ANSYS FLUENT 12.0

UDF Manual

April 2009

Contents

About This Document

User-defined functions (UDFs) allow you to customize **ANSYS FLUENT** and can significantly enhance its capabilities. This UDF Manual presents detailed information on how to write, compile, and use UDFs in **ANSYS FLUENT**. Examples have also been included, where available. General information about C programming basics is included in an appendix.

Information in this manual is presented in the following chapters:

This document provides some basic information about the C programming language (Appendix A) as it relates to user-defined functions in ANSYS FLUENT, and assumes that you are an experienced programmer in C. If you are unfamiliar with C, please consult a C language reference guide (e.g., [6, 9]) before you begin the process of writing UDFs and using them in your ANSYS FLUENT model.

This document does not imply responsibility on the part of ANSYS, Inc. for the accuracy or stability of solutions obtained using UDFs that are either user-generated or provided by ANSYS, Inc. Support for current license holders will be limited to guidance related to communication between a UDF and the ANSYS FLUENT solver. Other aspects of the UDF development process that include conceptual function design, implementation (writing C code), compilation and debugging of C source code, execution of the UDF, and function design verification will remain the responsibility of the UDF author.

UDF compiled libraries are specific to the computer architecture being used and the version of the ANSYS FLUENT executable being run and must be rebuilt any

time ANSYS FLUENT is upgraded, your operating system changes, or the job is run on a different type of computer. Note that UDFs may need to be updated with new versions of ANSYS FLUENT.

Chapter 1. Overview

This chapter contains an overview of user-defined functions (UDFs) and their usage in ANSYS FLUENT. Details about UDF functionality are described in the following sections:

- Section 1.1: What is a User-Defined Function (UDF)?

- Section 1.2: Why Use UDFs?

- Section 1.3: Limitations

- Section 1.4: Defining Your UDF Using DEFINE Macros

- Section 1.5: Interpreting and Compiling UDFs

- Section 1.6: Hooking UDFs to Your ANSYS FLUENT Model

- Section 1.7: Mesh Terminology

- Section 1.8: Data Types in ANSYS FLUENT

- Section 1.9: UDF Calling Sequence in the Solution Process

- Section 1.10: Special Considerations for Multiphase UDFs

1.1 What is a User-Defined Function (UDF)?

A user-defined function, or UDF, is a function that you program that can be dynamically loaded with the ANSYS FLUENT solver to enhance the standard features of the code. For example, you can use a UDF to define your own boundary conditions, material properties, and source terms for your flow regime, as well as specify customized model parameters (e.g., DPM, multiphase models), initialize a solution, or enhance postprocessing. See Section 1.2: Why Use UDFs? for more examples.

UDFs are written in the C programming language using any text editor and the source code file is saved with a `.c` extension (e.g., `myudf.c`). One source file can contain a single UDF or multiple UDFs, and you can define multiple source files. See Appendix A for some basic information on C programming.

UDFs are defined using `DEFINE` macros provided by ANSYS FLUENT (see Chapter 2: `DEFINE` Macros). They are coded using additional macros and functions also supplied by ANSYS FLUENT that access ANSYS FLUENT solver data and perform other tasks. See Chapter 3: Additional Macros for Writing UDFs for details.

Every UDF must contain the `udf.h` file inclusion directive (`#include "udf.h"`) at the beginning of the source code file, which allows definitions of `DEFINE` macros and other ANSYS FLUENT-provided macros and functions to be included during the compilation process. See Section 1.4.1: Including the `udf.h` Header File in Your Source File for details. Note that values that are passed to a solver by a UDF or returned by the solver to a UDF are specified in SI units.

Source files containing UDFs can be either interpreted or compiled in ANSYS FLUENT. For interpreted UDFs, source files are interpreted and loaded directly at *runtime*, in a single-step process. For compiled UDFs, the process involves two separate steps. A shared object code library is first built and then it is loaded into ANSYS FLUENT. See Chapter 4: Interpreting UDFs and Chapter 5: Compiling UDFs. After being interpreted or compiled, UDFs will become visible and selectable in ANSYS FLUENT dialog boxes, and can be hooked to a solver by choosing the function name in the appropriate dialog box. This process is described in Chapter 6: Hooking UDFs to ANSYS FLUENT.

In summary, UDFs:

- are written in the C programming language. (Appendix A)

- must have an include statement for the `udf.h` file. (Section 1.4.1: Including the `udf.h` Header File in Your Source File)

- must be defined using `DEFINE` macros supplied by ANSYS FLUENT (Chapter 2: `DEFINE` Macros)

- utilize predefined macros and functions supplied by ANSYS FLUENT to access ANSYS FLUENT solver data and perform other tasks. (Chapter 3: Additional Macros for Writing UDFs)

- are executed as interpreted or compiled functions. (Chapter 4: Interpreting UDFs and Chapter 5: Compiling UDFs)

- are hooked to an ANSYS FLUENT solver using a graphical user interface dialog box. (Chapter 6: Hooking UDFs to ANSYS FLUENT)

- use and return values specified in SI units.

1.2 Why Use UDFs?

UDFs allow you to customize ANSYS FLUENT to fit your particular modeling needs. UDFs can be used for a variety of applications. Some examples include

- customization of boundary conditions, material property definitions, surface and volume reaction rates, source terms in ANSYS FLUENT transport equations, source terms in user-defined scalar (UDS) transport equations, diffusivity functions, etc.

- adjustment of computed values on a once-per-iteration basis

- initialization of a solution

- asynchronous (on demand) execution of a UDF

- execution at the end of an iteration, upon exit from ANSYS FLUENT, or upon loading of a compiled UDF library

- postprocessing enhancement

- enhancement of existing ANSYS FLUENT models (e.g., discrete phase model, multiphase mixture model, discrete ordinates radiation model)

Simple examples of UDFs that demonstrate usage are provided with most DEFINE macro descriptions in Chapter 2: DEFINE Macros. In addition, a step-by-step example (minitutorial) and detailed examples can be found in Chapter 8: Examples.

1.3 Limitations

Although the UDF capability in ANSYS FLUENT can address a wide range of applications, it is not possible to address every application using UDFs. Not all solution variables or ANSYS FLUENT models can be accessed by UDFs. If you are unsure whether a particular problem can be handled using a UDF, contact your technical support engineer for assistance.

$\boxed{i}$ Note that you may need to update your UDF when using a new version of ANSYS FLUENT.

1.4 Defining Your UDF Using DEFINE Macros

UDFs are defined using ANSYS FLUENT-supplied function declarations. These function declarations are implemented in the code as macros, and are referred to in this document as DEFINE (all capitals) macros. Definitions for DEFINE macros are contained in the udf.h header file (see Appendix B for a listing). For a complete description of each DEFINE macro and an example of its usage, refer to Chapter 2: DEFINE Macros.

The general format of a DEFINE macro is

```
DEFINE_MACRONAME(udf_name, passed-in variables)
```

where the first argument in the parentheses is the name of the UDF that you supply. Name arguments are case-sensitive and *must* be specified in lowercase. The name that you choose for your UDF will become visible and selectable in drop-down lists in graphical user interface dialog boxes in ANSYS FLUENT, after the function has been interpreted or compiled. The second set of input arguments to the DEFINE macro are variables that are passed into your function from the ANSYS FLUENT solver.

For example, the macro

```
DEFINE_PROFILE(inlet_x_velocity, thread, index)
```

defines a boundary profile function named inlet_x_velocity with two variables, thread and index, that are passed into the function from ANSYS FLUENT. These passed-in variables are the boundary condition zone ID (as a pointer to the thread) and the index identifying the variable that is to be stored. After the UDF has been interpreted or compiled, its name (e.g., inlet_x_velocity) will become visible and selectable in drop-down lists in the appropriate boundary condition dialog box (e.g., Velocity Inlet) in ANSYS FLUENT.

> i Note that all of the arguments to a DEFINE macro need to be placed on the same line in your source code. Splitting the DEFINE statement onto several lines will result in a compilation error.

> i Make sure that there are no spaces between the macro (e.g., DEFINE_PROFILE) and the first parenthesis of the arguments, as this will cause an error in Windows.

> i Do not include a DEFINE macro statement (e.g., DEFINE_PROFILE) within a comment in your source code. This will cause a compilation error.

1.4.1 Including the udf.h Header File in Your Source File

The udf.h header file contains definitions for DEFINE macros as well as #include compiler directives for C library function header files. It also includes header files (e.g., mem.h) for other ANSYS FLUENT-supplied macros and functions. You must, therefore, include the udf.h file at the beginning of *every* UDF source code file using the #include compiler directive:

```
#include "udf.h"
```

For example, when udf.h is included in the source file containing the DEFINE statement from the previous section,

```
#include "udf.h"

DEFINE_PROFILE(inlet_x_velocity, thread, index)
```

upon compilation, the macro will expand to

```
void inlet_x_velocity(Thread *thread, int index)
```

> *i* You do not need to put a copy of udf.h in your local folder when you compile your UDF. The ANSYS FLUENT solver automatically reads the udf.h file from the following folder after your UDF is compiled:
>
> $$path\backslash \texttt{ANSYS Inc}\backslash \texttt{v120}\backslash \texttt{fluent}\backslash \texttt{fluent12.0.}\overset{\Downarrow}{x}\ \backslash \texttt{src}\backslash$$
>
> where *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is C:\Program Files), and x is replaced by the appropriate number for the release (e.g., 9 for fluent12.0.9).

1.5 Interpreting and Compiling UDFs

Source code files containing UDFs can be either interpreted or compiled in ANSYS FLU-ENT. In both cases the functions are compiled, but the way in which the source code is compiled, and the code that results from the compilation process is different for the two methods. These differences are explained below.

Compiled UDFs

Compiled UDFs are built in the same way that the ANSYS FLUENT executable itself is built. A script called `Makefile` is used to invoke the system C compiler to build an object code library. You initiate this action in the Compiled UDFs dialog box by clicking the Build button. The object code library contains the native machine language translation of your higher-level C source code. The shared library must then loaded into ANSYS FLUENT at runtime by a process called "dynamic loading." You initiate this action in the Compiled UDFs dialog box by clicking the Load button. The object libraries are specific to the computer architecture being used, as well as to the particular version of the ANSYS FLUENT executable being run. The libraries must, therefore, be rebuilt any time ANSYS FLUENT is upgraded, when the computer's operating system level changes, or when the job is run on a different type of computer.

In summary, compiled UDFs are compiled from source files using the graphical user interface, in a two-step process. The process involves the Compiled UDFs dialog box, where you first build a shared library object file from a source file, and then load the shared library that was just built into ANSYS FLUENT.

Interpreted UDFs

Interpreted UDFs are interpreted from source files using the graphical user interface, but in a single-step process. The process, which occurs at *runtime*, involves a visit to the Interpreted UDFs dialog box where you Interpret a source file.

Inside ANSYS FLUENT, the source code is compiled into an intermediate, architecture-independent machine code using a C preprocessor. This machine code then executes on an internal emulator, or interpreter, when the UDF is invoked. This extra layer of code incurs a performance penalty, but allows an interpreted UDF to be shared effortlessly between different architectures, operating systems, and ANSYS FLUENT versions. If execution speed does become an issue, an interpreted UDF can always be run in compiled mode without modification.

The interpreter that is used for interpreted UDFs does not have all of the capabilities of a standard C compiler (which is used for compiled UDFs). Specifically, interpreted UDFs *cannot* contain any of the following C programming language elements:

- goto statements

- non ANSI-C prototypes for syntax

- direct data structure references

- declarations of local structures

- unions

- pointers to functions

- arrays of functions

- multi-dimensional arrays

1.5.1 Differences Between Interpreted and Compiled UDFs

The major difference between interpreted and compiled UDFs is that interpreted UDFs cannot access ANSYS FLUENT solver data using direct structure references; they can only indirectly access data through the use of ANSYS FLUENT-supplied macros. This can be significant if, for example, you want to introduce new data structures in your UDF.

A summary of the differences between interpreted and compiled UDFs is presented below. See Chapters 4 and 5 for details on interpreting and compiling UDFs, respectively, in ANSYS FLUENT.

- Interpreted UDFs

    - are portable to other platforms.

    - can all be run as compiled UDFs.

    - do not require a C compiler.

    - are slower than compiled UDFs.

    - are restricted in the use of the C programming language.

    - cannot be linked to compiled system or user libraries.

    - can access data stored in an ANSYS FLUENT structure *only* using a predefined macro (see Chapters 3).

- Compiled UDFs

    - execute faster than interpreted UDFs.

    - are *not* restricted in the use of the C programming language.

    - can call functions written in other languages (specifics are system- and compiler-dependent).

    - cannot necessarily be run as interpreted UDFs if they contain certain elements of the C language that the interpreter cannot handle.

In summary, when deciding which type of UDF to use for your ANSYS FLUENT model

- use interpreted UDFs for small, straightforward functions.

- use compiled UDFs for complex functions that

    - have a significant CPU requirement (e.g., a property UDF that is called on a per-cell basis every iteration).

    - require access to a shared library.

1.6 Hooking UDFs to Your ANSYS FLUENT Model

After your UDF source file is interpreted or compiled, the function(s) contained in the interpreted code or shared library will appear in drop-down lists in graphical user interface dialog boxes, ready for you to activate or "hook" to your CFD model. See Chapter 6: Hooking UDFs to ANSYS FLUENT for details on how to hook a UDF to ANSYS FLUENT.

1.7 Mesh Terminology

Most user-defined functions access data from an ANSYS FLUENT solver. Since solver data is defined in terms of mesh components, you will need to learn some basic mesh terminology before you can write a UDF.

A mesh is broken up into control volumes, or cells. Each cell is defined by a set of nodes, a cell center, and the faces that bound the cell (Figure 1.7.1). ANSYS FLUENT uses internal data structures to define the domain(s) of the mesh, to assign an order to cells, cell faces, and nodes in a mesh, and to establish connectivity between adjacent cells.

A thread is a data structure in ANSYS FLUENT that is used to store information about a boundary or cell zone. Cell threads are groupings of cells, and face threads are groupings of faces. Pointers to thread data structures are often passed to functions and manipulated in ANSYS FLUENT to access the information about the boundary or cell zones represented by each thread. Each boundary or cell zone that you define in your ANSYS FLUENT model in a boundary conditions dialog box has an integer Zone ID that is associated with the data contained within the zone. You will not see the term "thread" in a graphics dialog box in ANSYS FLUENT so you can think of a 'zone' as being the same as a 'thread' data structure when programming UDFs.

Cells and cell faces are grouped into zones that typically define the physical components of the model (e.g., inlets, outlets, walls, fluid regions). A face will bound either one or two cells depending on whether it is a boundary face or an interior face. A domain is a data structure in ANSYS FLUENT that is used to store information about a collection of node, face threads, and cell threads in a mesh.
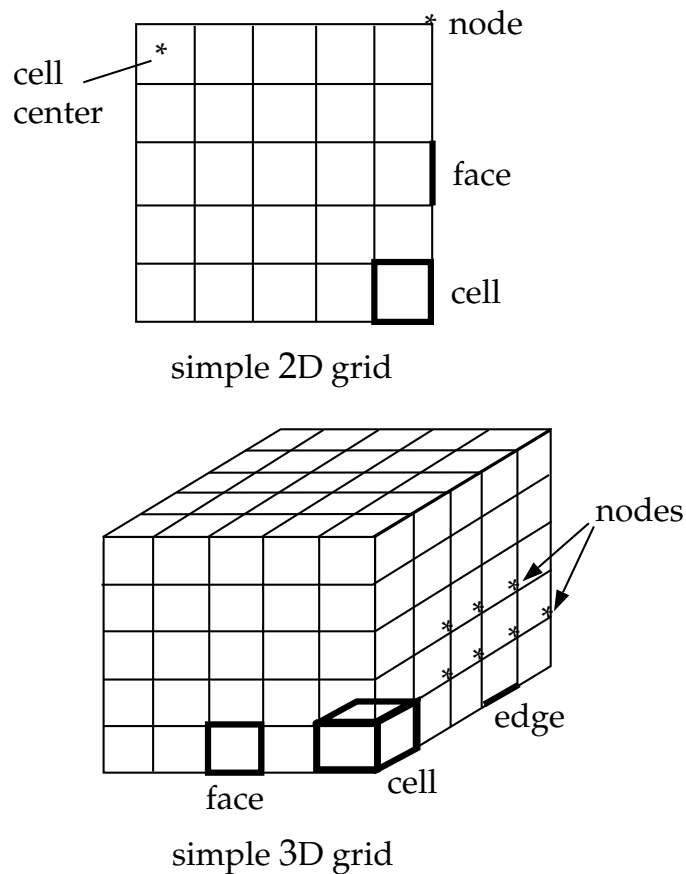
Figure 1.7.1: Mesh Components

node	mesh point
node thread	grouping of nodes
edge	boundary of a face (3D)
face	boundary of a cell (2D or 3D)
face thread	grouping of faces
cell	control volume into which domain is broken up
cell center	location where cell data is stored
cell thread	grouping of cells
domain	a grouping of node, face, and cell threads

1.8 Data Types in ANSYS FLUENT

In addition to standard C language data types such as `real`, `int`, etc. that can be used to define data in your UDF, there are ANSYS FLUENT-specific data types that are associated with solver data. These data types represent the computational units for a mesh (Figure 1.7.1). Variables that are defined using these data types are typically supplied as arguments to `DEFINE` macros as well as to other special functions that access ANSYS FLUENT solver data.

Some of the more commonly used ANSYS FLUENT data types are:

```
Node
face_t
cell_t
Thread
Domain
```

`Node` is a structure data type that stores data associated with a mesh point.

`face_t` is an integer data type that identifies a particular face within a face thread.

`cell_t` is an integer data type that identifies a particular cell within a cell thread.

`Thread` is a structure data type that stores data that is common to the group of cells or faces that it represents. For multiphase applications, there is a thread structure for each phase, as well as for the mixture. See Section 1.10.1: Multiphase-specific Data Types for details.

`Domain` is a structure data type that stores data associated with a collection of node, face, and cell threads in a mesh. For single-phase applications, there is only a single domain structure. For multiphase applications, there are domain structures for each phase, the interaction between phases, as well as for the mixture. The mixture-level domain is the highest-level structure for a multiphase model. See Section 1.10.1: Multiphase-specific Data Types for details.

i Note that all of the ANSYS FLUENT data types are case sensitive.

When you use a UDF in ANSYS FLUENT, your function can access solution variables at individual cells or cell faces in the fluid and boundary zones. UDFs need to be passed appropriate arguments such as a thread reference (i.e., pointer to a particular thread) and the cell or face ID in order to allow individual cells or faces to be accessed. Note that a face ID or cell ID by itself does not uniquely identify the face or cell. A thread pointer is always required along with the ID to identify which thread the face (or cell) belongs to.

Some UDFs are passed the cell index variable (`c`) as an argument (such as in `DEFINE_PROPERTY(my_function,c,t)`), or the face index variable (`f`) (such as in `DEFINE_UDS_FLUX(my_function,f,t,i)`). If the cell or face index variable (e.g., `cell_t c`, `face_t f`) is not passed as an argument and is needed in the UDF, the variable is always available to be used by the function after it has been declared locally. See Section 2.7.3: `DEFINE_UDS_FLUX` for an example.

The data structures that are passed to your UDF (as pointers) depend on the `DEFINE` macro you are using and the property or term you are trying to modify. For example, `DEFINE_ADJUST` UDFs are general-purpose functions that are passed a domain pointer (`d`) (such as in `DEFINE_ADJUST(my_function, d)`). `DEFINE_PROFILE` UDFs are passed a thread pointer (`t`) to the boundary zone that the function is hooked to, such as in `DEFINE_PROFILE(my_function, thread, i)`.

Some UDFs (such as `DEFINE_ON_DEMAND` functions) are not passed any pointers to data structures, while others are not passed the pointer the UDF needs. If your UDF needs to access a thread or domain pointer that is not *directly* passed by the solver through an argument, then you will need to use a special ANSYS FLUENT-supplied macro to obtain the pointer in your UDF. For example, `DEFINE_ADJUST` is passed only the domain pointer, so if your UDF needs a thread pointer, it will have to declare the variable locally and then obtain it using the special macro `Lookup_Thread`. An exception to this is if your UDF needs a thread pointer to loop over all of the cell threads or all the face threads in a domain (using `thread_c_loop(c,t)` or `thread_f_loop(f,t)`, respectively) and it is not passed to the `DEFINE` macro. Since the UDF will be looping over all threads in the domain, you will not need to use `Lookup_Thread` to get the thread pointer to pass it to the looping macro; you'll just need to declare the thread pointer (and cell or face ID) locally before calling the loop. See Section 2.2.1: `DEFINE_ADJUST` for an example.

As another example, if you are using `DEFINE_ON_DEMAND` (which is not passed any pointer argument) to execute an asynchronous UDF and your UDF needs a domain pointer, then the function will need to declare the domain variable locally and obtain it using `Get_Domain`. See Section 2.2.9: `DEFINE_ON_DEMAND` for an example. Refer to Section 3.2.6: Special Macros for details.

1.9   UDF Calling Sequence in the Solution Process

UDFs are called at predetermined times in the ANSYS FLUENT solution process. However, they can also be executed asynchronously (or "on demand") using a DEFINE_ON_DEMAND UDF. If a DEFINE_EXECUTE_AT_END UDF is utilized, then ANSYS FLUENT calls the function at the end of an iteration. A DEFINE_EXECUTE_AT_EXIT is called at the end of an ANSYS FLUENT session while a DEFINE_EXECUTE_ON_LOADING is called whenever a UDF compiled library is loaded. Understanding the context in which UDFs are called within ANSYS FLUENT's solution process may be important when you begin the process of writing UDF code, depending on the type of UDF you are writing. The solver contains call-outs that are linked to user-defined functions that you write. Knowing the sequencing of function calls within an iteration in the ANSYS FLUENT solution process can help you determine which data are current and available at any given time.

Pressure-Based Segregated Solver

The solution process for the pressure-based segregated solver (Figure 1.9.1) begins with a two-step initialization sequence that is executed outside the solution iteration loop. This sequence begins by initializing equations to user-entered (or default) values taken from the ANSYS FLUENT user interface. Next, PROFILE UDFs are called, followed by a call to INIT UDFs. Initialization UDFs overwrite initialization values that were previously set.

The solution iteration loop begins with the execution of ADJUST UDFs. Next, momentum equations for u, v, and w velocities are solved sequentially, followed by mass continuity and velocity updates. Subsequently, the energy and species equations are solved, followed by turbulence and other scalar transport equations, as required. Note that PROFILE and SOURCE UDFs are called by each "Solve" routine for the variable currently under consideration (e.g., species, velocity).

After the conservation equations, properties are updated, including PROPERTY UDFs. Thus, if your model involves the gas law, for example, the density will be updated at this time using the updated temperature (and pressure and/or species mass fractions). A check for either convergence or additional requested iterations is done, and the loop either continues or stops.

Pressure-Based Coupled Solver

The solution process for the pressure-based coupled solver (Figure 1.9.2) begins with a two-step initialization sequence that is executed outside the solution iteration loop. This sequence begins by initializing equations to user-entered (or default) values taken from the ANSYS FLUENT user interface. Next, PROFILE UDFs are called, followed by a call to INIT UDFs. Initialization UDFs overwrite initialization values that were previously set.

The solution iteration loop begins with the execution of ADJUST UDFs. Next, ANSYS FLUENT solves the governing equations of continuity and momentum in a coupled fashion, which is simultaneously as a set, or vector, of equations. Energy, species transport, turbulence, and other transport equations as required are subsequently solved sequentially, and the remaining process is the same as the pressure-based segregated solver.

Density-Based Solver

As is the case for the other solvers, the solution process for the density-based solver (Figure 1.9.3) begins with a two-step initialization sequence that is executed outside the solution iteration loop. This sequence begins by initializing equations to user-entered (or default) values taken from the ANSYS FLUENT user interface. Next, PROFILE UDFs are called, followed by a call to INIT UDFs. Initialization UDFs overwrite initialization values that were previously set.

The solution iteration loop begins with the execution of ADJUST UDFs. Next, ANSYS FLUENT solves the governing equations of continuity and momentum, energy, and species transport in a coupled fashion, which is simultaneously as a set, or vector, of equations. Turbulence and other transport equations as required are subsequently solved sequentially, and the remaining process is the same as the pressure-based segregated solver.
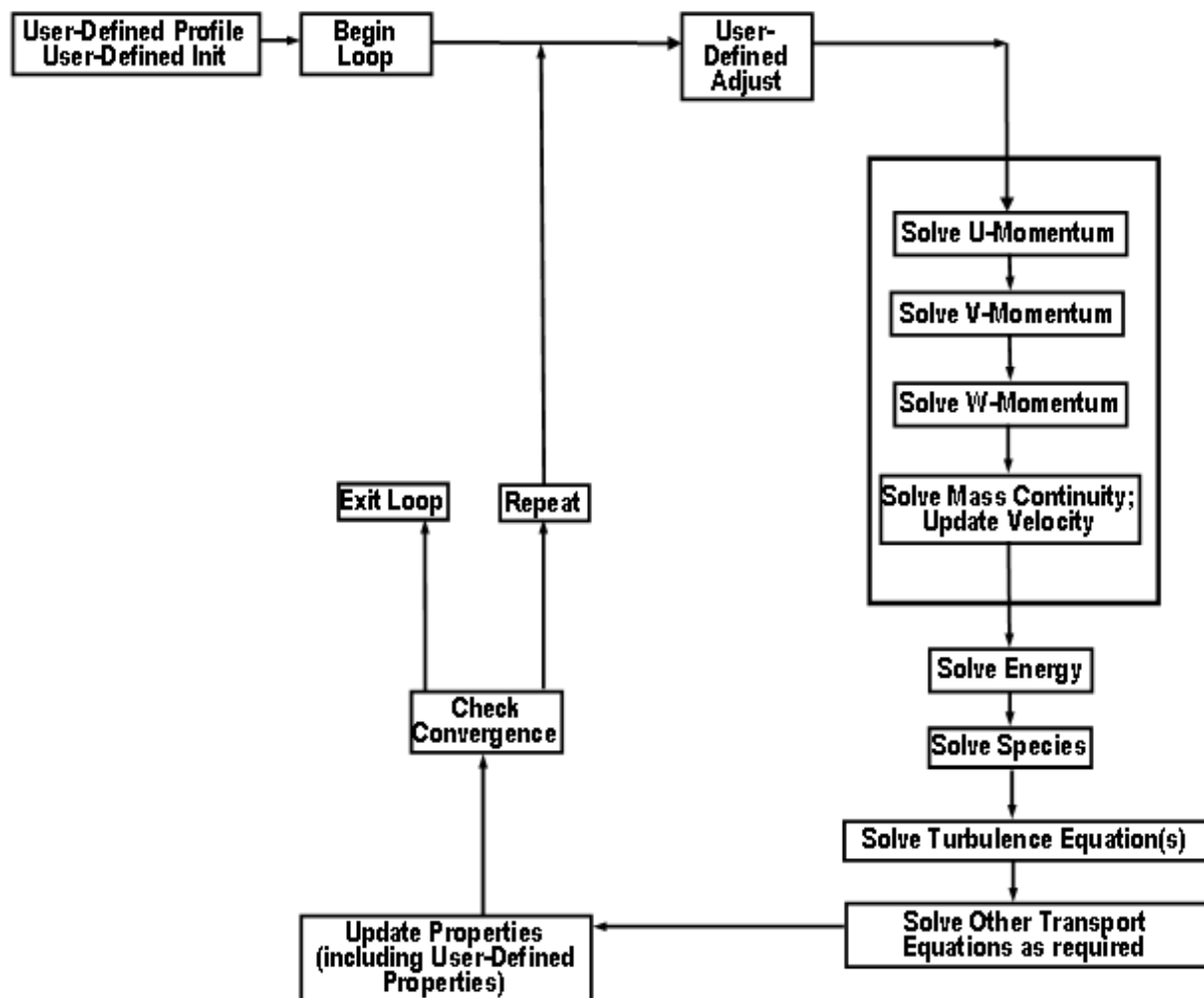
Figure 1.9.1: Solution Procedure for the Pressure-Based Segregated Solver
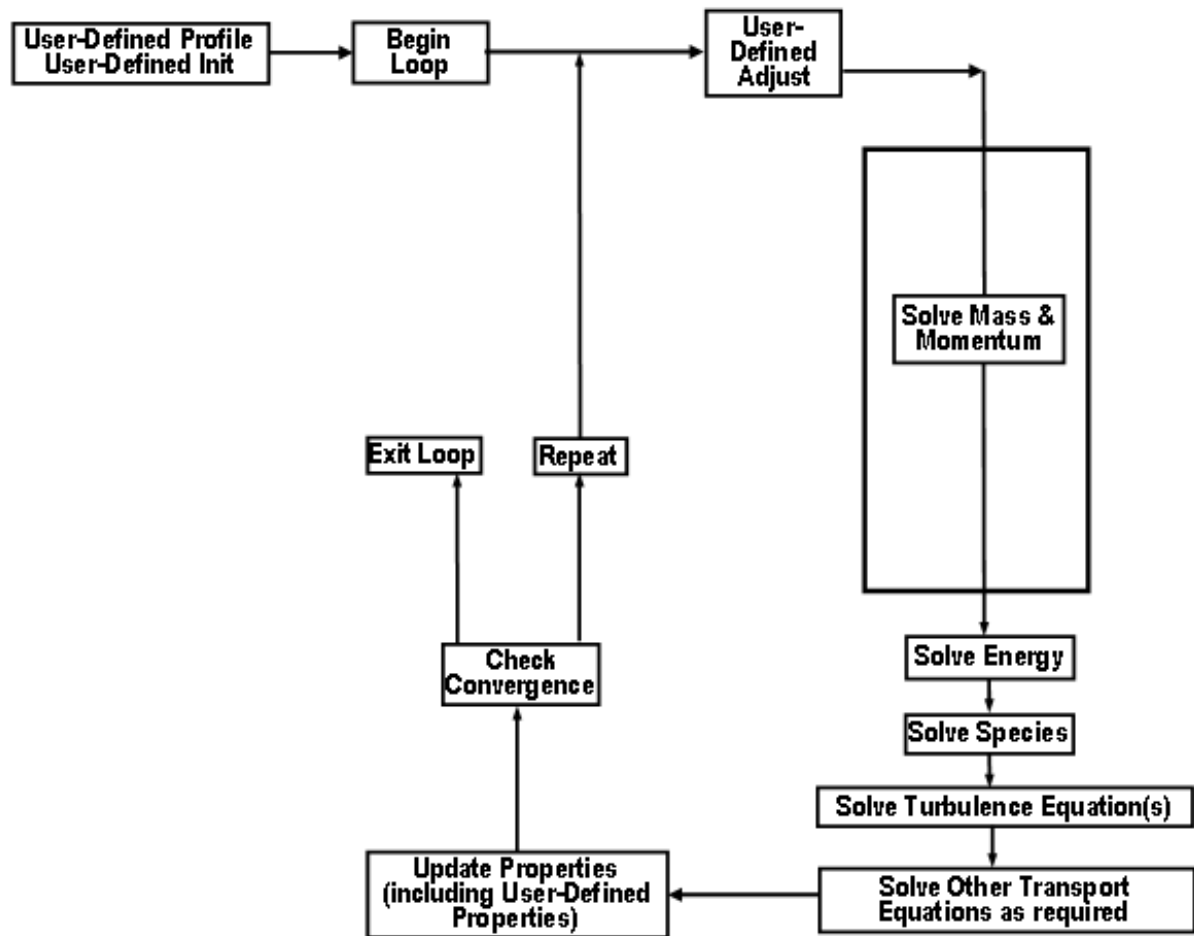
Figure 1.9.2: Solution Procedure for the Pressure-Based Coupled Solver
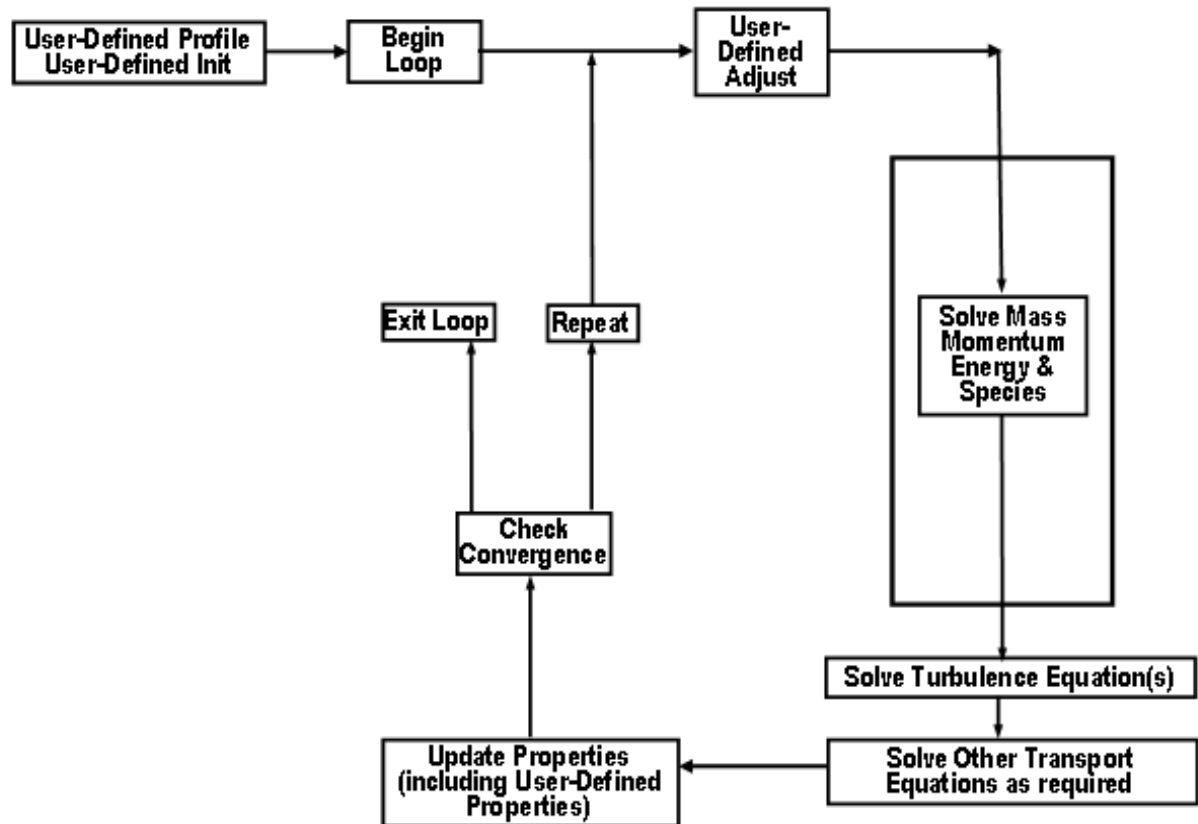
Figure 1.9.3: Solution Procedure for the Density-Based Solver

1.10   Special Considerations for Multiphase UDFs

In many cases, the UDF source code that you will write for a single-phase flow will be the same as for a multiphase flow. For example, there will be no differences between the C code for a single-phase boundary profile (defined using DEFINE_PROFILE) and the code for a multiphase profile, assuming that the function is accessing data *only* from the phase-level domain to which it is hooked in the graphical user interface. If your UDF is *not* explicitly passed a pointer to the thread or domain structure that it requires, you will need to use a special multiphase-specific macro (e.g., THREAD_SUB_THREAD) to retrieve it. This is discussed in Chapter 3: Additional Macros for Writing UDFs.

See Appendix B for a complete list of general-purpose DEFINE macros and multiphase-specific DEFINE macros that can be used to define UDFs for multiphase model cases.

1.10.1   Multiphase-specific Data Types

In addition to the ANSYS FLUENT-specific data types presented in Section 1.8: Data Types in ANSYS FLUENT, there are special thread and domain data structures that are specific to multiphase UDFs. These data types are used to store properties and variables for the mixture of all of the phases, as well as for each individual phase when a multiphase model (i.e., Mixture, VOF, Eulerian) is used.

In a multiphase application, the top-level domain is referred to as the 'superdomain'. Each phase occupies a domain referred to as a 'subdomain'. A third domain type, the 'interaction' domain, is introduced to allow for the definition of phase interaction mechanisms. When mixture properties and variables are needed (a sum over phases), the superdomain is used for those quantities, while the subdomain carries the information for individual phases. In single-phase, the concept of a mixture is used to represent the sum over all the species (components), while in multiphase it represents the sum over all the phases. This distinction is important, because ANSYS FLUENT has the capability of handling multiphase multi-components, where, for example, a phase can consist of a mixture of species.

Since solver information is stored in thread data structures, threads must be associated with the superdomain as well as with each of the subdomains. In other words, for each cell or face thread defined in the superdomain, there is a corresponding cell or face thread defined for each subdomain. Some of the information defined in one thread of the superdomain is shared with the corresponding threads of each of the subdomains. Threads associated with the superdomain are referred to as 'superthreads', while threads associated with the subdomain are referred to as phase-level threads, or 'subthreads'. The domain and thread hierarchy are summarized in Figure 1.10.1.
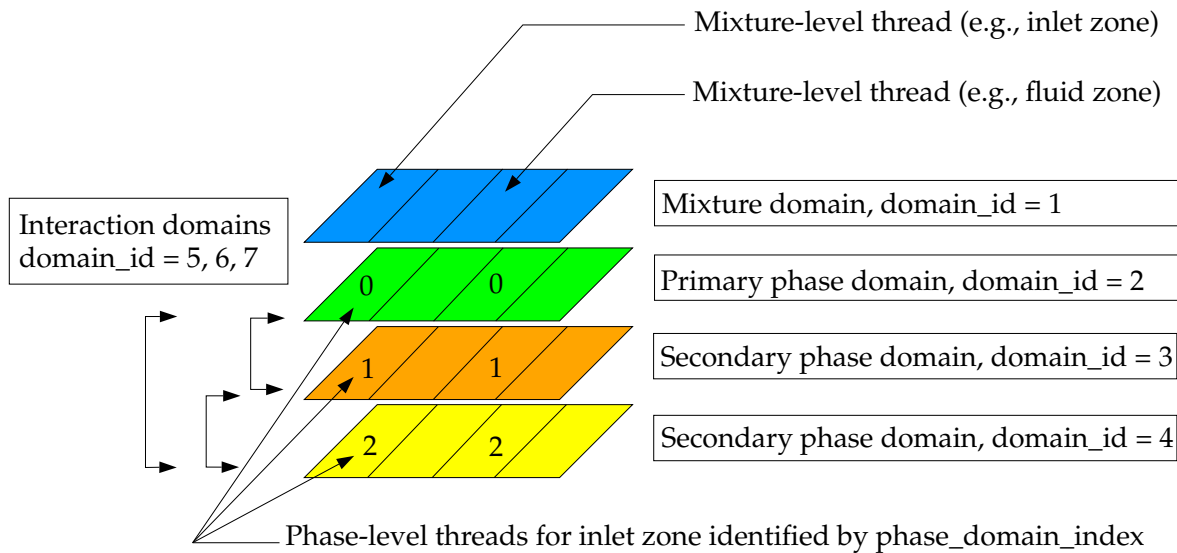
Figure 1.10.1: Domain and Thread Structure Hierarchy

Figure 1.10.1 introduces the concept of the `domain_id` and `phase_domain_index`. The `domain_id` can be used in UDFs to distinguish the superdomain from the primary and secondary phase-level domains. The superdomain (mixture domain) `domain_id` is always assigned the value of 1. Interaction domains are also identified with the `domain_id`. The `domain_id`s are not necessarily ordered sequentially as shown in Figure 1.10.1.

The `phase_domain_index` can be used in UDFs to distinguish between the primary and secondary phase-level threads. `phase_domain_index` is always assigned the value of 0 for the primary phase-level thread.

The data structures that are passed to a UDF depend on the multiphase model that is enabled, the property or term that is being modified, the `DEFINE` macro that is used, and the domain that is to be affected (mixture or phase). To better understand this, consider the differences between the Mixture and Eulerian multiphase models. In the Mixture model, a single momentum equation is solved for a mixture whose properties are determined from the sum of its phases. In the Eulerian model, a momentum equation is solved for each phase. ANSYS FLUENT allows you to directly specify a momentum source for the mixture of phases (using `DEFINE_SOURCE`) when the mixture model is used, but not for the Eulerian model. For the latter case, you can specify momentum sources for the individual phases. Hence, the multiphase model, as well as the term being modified by the UDF, determines which domain or thread is required.

UDFs that are hooked to the mixture of phases are passed superdomain (or mixture-level) structures, while functions that are hooked to a particular phase are passed subdomain (or phase-level) structures. `DEFINE_ADJUST` and `DEFINE_INIT` UDFs are hardwired to the mixture-level domain. Other types of UDFs are hooked to different phase domains. For your convenience, Appendix B contains a list of multiphase models in ANSYS FLUENT and the phase on which UDFs are specified for the given variables. From this information, you can infer which domain structure is passed from the solver to the UDF.

Chapter 2.   `DEFINE` **Macros**

This chapter contains descriptions of predefined `DEFINE` macros that you will use to define your UDF.

The chapter is organized in the following sections:

- Section 2.1: Introduction

- Section 2.2: General Purpose `DEFINE` Macros

- Section 2.3: Model-Specific `DEFINE` Macros

- Section 2.4: Multiphase `DEFINE` Macros

- Section 2.5: Discrete Phase Model (DPM) `DEFINE` Macros

- Section 2.6: Dynamic Mesh `DEFINE` Macros

- Section 2.7: User-Defined Scalar (UDS) Transport Equation `DEFINE` Macros

2.1   Introduction

`DEFINE` macros are predefined macros provided by ANSYS, Inc. that must be used to define your UDF. A listing and discussion of each `DEFINE` macros is presented below. (Refer to Section 1.4: Defining Your UDF Using `DEFINE` Macros for general information about `DEFINE` macros.) Definitions for `DEFINE` macros are contained within the `udf.h` file. For your convenience, they are provided in Appendix B.

For each of the `DEFINE` macros listed in this chapter, a source code example of a UDF that utilizes it is provided, where available. Many of the examples make extensive use of other macros presented in Chapter 3: Additional Macros for Writing UDFs. Note that not all of the examples in the chapter are complete functions that can be executed as stand-alone UDFs in ANSYS FLUENT. Examples are intended to demonstrate `DEFINE` macro usage only.

Special care must be taken for some serial UDFs that will be run in parallel ANSYS FLUENT. See Chapter 7: Parallel Considerations for details.

> **i** Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

i Make sure that there are no spaces between the macro (e.g., DEFINE␣PROFILE) and the first parenthesis of the arguments, as this will cause an error in Windows.

i Do not include a DEFINE macro statement (e.g., DEFINE␣PROFILE) within a comment in your source code. This will cause a compilation error.

2.2 General Purpose DEFINE Macros

The DEFINE macros presented in this section implement general solver functions that are independent of the model(s) you are using in ANSYS FLUENT. Table 2.2.1 provides a quick reference guide to these DEFINE macros, the functions they are used to define, and the dialog boxes where they are activated or "hooked" to ANSYS FLUENT. Definitions of each DEFINE macro are contained in udf.h can be found in Appendix B.

- Section 2.2.1: DEFINE_ADJUST

- Section 2.2.2: DEFINE_DELTAT

- Section 2.2.3: DEFINE_EXECUTE_AT_END

- Section 2.2.4: DEFINE_EXECUTE_AT_EXIT

- Section 2.2.5: DEFINE_EXECUTE_FROM_GUI

- Section 2.2.6: DEFINE_EXECUTE_ON_LOADING

- Section 2.2.7: DEFINE_EXECUTE_AFTER_CASE/DATA

- Section 2.2.8: DEFINE_INIT

- Section 2.2.9: DEFINE_ON_DEMAND

- Section 2.2.10: DEFINE_RW_FILE

Table 2.2.1: Quick Reference Guide for General Purpose DEFINE Macros

Function	DEFINE Macro	Dialog Box Activated In
manipulates variables	DEFINE_ADJUST	User-Defined Function Hooks
time step size (for time dependent solutions)	DEFINE_DELTAT	Adaptive Time Step Settings
executes at end of iteration	DEFINE_EXECUTE_AT_END	User-Defined Function Hooks
executes at end of an ANSYS FLUENT session	DEFINE_EXECUTE_AT_EXIT	User-Defined Function Hooks
executes from a user-defined Scheme routine	DEFINE_EXECUTE_FROM_GUI	N/A
executes when a UDF library is loaded	DEFINE_EXECUTE_ON_LOADING	N/A
executes after a case file is read	DEFINE_EXECUTE_AFTER_CASE	N/A
executes after a data file is read	DEFINE_EXECUTE_AFTER_DATA	N/A
initializes variables	DEFINE_INIT	User-Defined Function Hooks
executes asynchronously	DEFINE_ON_DEMAND	Execute On Demand
reads/writes variables to case and data files	DEFINE_RW_FILE	User-Defined Function Hooks

2.2.1 DEFINE_ADJUST

Description

DEFINE_ADJUST is a general-purpose macro that can be used to adjust or modify AN-SYS FLUENT variables that are *not* passed as arguments. For example, you can use DEFINE_ADJUST to modify flow variables (e.g., velocities, pressure) and compute integrals. You can also use it to integrate a scalar quantity over a domain and adjust a boundary condition based on the result. A function that is defined using DEFINE_ADJUST executes at every iteration and is called at the beginning of every iteration before transport equations are solved. For an overview of the ANSYS FLUENT solution process which shows when a DEFINE_ADJUST UDF is called, refer to Figures 1.9.1, 1.9.2, and 1.9.3.

Usage

DEFINE_ADJUST(name,d)

Argument Type	Description
symbol name	UDF name.
Domain *d	Pointer to the domain over which the adjust function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the pointer that is passed to the function by the solver is the mixture-level domain.

Function returns
void

There are two arguments to DEFINE_ADJUST: name and d. You supply name, the name of the UDF. d is passed by the ANSYS FLUENT solver to your UDF.

Example 1

The following UDF, named `my_adjust`, integrates the turbulent dissipation over the entire domain using `DEFINE_ADJUST`. This value is then displayed in the console. The UDF is called once every iteration. It can be executed as an interpreted or compiled UDF in ANSYS FLUENT.

```
/**********************************************************************
   UDF for integrating turbulent dissipation and displaying it in the
   console
**********************************************************************/

#include "udf.h"

DEFINE_ADJUST(my_adjust,d)
{
  Thread *t;
  /* Integrate dissipation. */
  real sum_diss=0.;
  cell_t c;

  thread_loop_c(t,d)
  {
   begin_c_loop(c,t)
       sum_diss += C_D(c,t)*
       C_VOLUME(c,t);
   end_c_loop(c,t)
  }

  printf("Volume integral of turbulent dissipation: %g\n", sum_diss);
}
```

Example 2

The following UDF, named adjust_fcn, specifies a user-defined scalar as a function of the gradient of another user-defined scalar, using DEFINE_ADJUST. The function is called once every iteration. It is executed as a compiled UDF in ANSYS FLUENT.

```
/************************************************************************
   UDF for defining user-defined scalars and their gradients
*************************************************************************/

#include "udf.h"

DEFINE_ADJUST(adjust_fcn,d)
{
  Thread *t;
  cell_t c;
  real K_EL = 1.0;

  /* Do nothing if gradient isn't allocated yet. */
  if (! Data_Valid_P())
    return;

  thread_loop_c(t,d)
    {
    if (FLUID_THREAD_P(t))
      {
       begin_c_loop_all(c,t)
          {
           C_UDSI(c,t,1) +=
                      K_EL*NV_MAG2(C_UDSI_G(c,t,0))*C_VOLUME(c,t);
          }
       end_c_loop_all(c,t)
      }
    }
}
```

Hooking an Adjust UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_ADJUST is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., adjust_fcn) will become visible and selectable via the User-Defined Function Hooks dialog box in ANSYS FLUENT. Note that you can hook multiple adjust functions to your model. See Section 6.1.1: Hooking DEFINE_ADJUST UDFs for details.

2.2.2 DEFINE_DELTAT

Description

DEFINE_DELTAT is a general-purpose macro that you can use to control the size of the time step during the solution of a transient problem. Note that this macro can be used only if Adaptive is selected from the Time Stepping Method drop-down list in the Run Calculation task page in ANSYS FLUENT.

Usage

DEFINE_DELTAT(name,d)

Argument Type	Description
symbol name	UDF name.
Domain *d	Pointer to domain over which the time stepping control function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the pointer that is passed to the function by the solver is the mixture-level domain.

Function returns
real

There are two arguments to DEFINE_DELTAT: name and domain. You supply name, the name of the UDF. domain is passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the real value of the physical time step and return it to the solver.

Example

The following UDF, named mydeltat, is a simple function that shows how you can use DEFINE_DELTAT to change the value of the time step in a simulation. First, CURRENT_TIME is used to get the value of the current simulation time (which is assigned to the variable flow_time). Then, for the first 0.5 seconds of the calculation, a time step of 0.1 is set. A time step of 0.2 is set for the remainder of the simulation. The time step variable is then returned to the solver. See Section 3.5: Time-Dependent Macros for details on CURRENT_TIME.

```
/************************************************************************
   UDF that changes the time step value for a time-dependent solution
************************************************************************/
#include "udf.h"

DEFINE_DELTAT(mydeltat,d)
{
   real time_step;
   real flow_time = CURRENT_TIME;
   if (flow_time < 0.5)
      time_step = 0.1;
   else
      time_step = 0.2;
   return time_step;
}
```

Hooking an Adaptive Time Step UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DELTAT is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g,. mydeltat) will become visible and selectable in the Adaptive Time Step Settings dialog box in ANSYS FLUENT. See Section 6.1.2: Hooking DEFINE_DELTAT UDFs for details.

2.2.3 DEFINE_EXECUTE_AT_END

Description

DEFINE_EXECUTE_AT_END is a general-purpose macro that is executed at the end of an iteration in a steady state run, or at the end of a time step in a transient run. You can use DEFINE_EXECUTE_AT_END when you want to calculate flow quantities at these particular times. Note that you do not have to specify whether your execute-at-end UDF gets executed at the end of a time step or the end of an iteration. This is done automatically when you select the steady or unsteady time method in your ANSYS FLUENT model.

Usage

```
DEFINE_EXECUTE_AT_END(name)
```

Argument Type   **Description**
`symbol name`   UDF name.

Function returns
`void`

There is only one argument to `DEFINE_EXECUTE_AT_END`: `name`. You supply `name`, the name of the UDF. Unlike `DEFINE_ADJUST`, `DEFINE_EXECUTE_AT_END` is not passed a domain pointer. Therefore, if your function requires access to a domain pointer, then you will need to use the utility `Get_Domain(ID)` to explicitly obtain it (see Section 3.2.6: Domain Pointer (`Get_Domain`) and the example below). If your UDF requires access to a phase domain pointer in a multiphase solution, then it will need to pass the appropriate phase ID to `Get_Domain` in order to obtain it.

Example

The following UDF, named `execute_at_end`, integrates the turbulent dissipation over the entire domain using `DEFINE_EXECUTE_AT_END` and displays it in the console at the end of the current iteration or time step. It can be executed as an interpreted or compiled UDF in ANSYS FLUENT.

```
/***********************************************************************
   UDF for integrating turbulent dissipation and displaying it in the
   console at the end of the current iteration or time step
***********************************************************************/

#include "udf.h"

DEFINE_EXECUTE_AT_END(execute_at_end)
{

  Domain *d;
  Thread *t;
  /* Integrate dissipation. */
  real sum_diss=0.;
  cell_t c;
  d = Get_Domain(1);   /* mixture domain if multiphase */

 thread_loop_c(t,d)
```

```
    {
     if (FLUID_THREAD_P(t))
       {
        begin_c_loop(c,t)
             sum_diss += C_D(c,t) * C_VOLUME(c,t);
        end_c_loop(c,t)
       }
    }

   printf("Volume integral of turbulent dissipation: %g\n", sum_diss);
   fflush(stdout);
}
```

Hooking an Execute-at-End UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EXECUTE_AT_END is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g. execute_at_end) will become visible and selectable via the User-Defined Function Hooks dialog box in ANSYS FLUENT. Note that you can hook multiple end-iteration functions to your model. See Section 6.1.3: Hooking DEFINE_EXECUTE_AT_END UDFs for details.

2.2.4 DEFINE_EXECUTE_AT_EXIT

Description

DEFINE_EXECUTE_AT_EXIT is a general-purpose macro that can be used to execute a function at the end of an ANSYS FLUENT session.

Usage

DEFINE_EXECUTE_AT_EXIT(name)

Argument Type	Description
symbol name	UDF name.

Function returns
void

There is only one argument to DEFINE_EXECUTE_AT_EXIT: name. You supply name, the name of the UDF.

Hooking an Execute-at-Exit UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EXECUTE_AT_EXIT is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable via the User-Defined Function Hooks dialog box in ANSYS FLUENT. Note that you can hook multiple at-exit UDFs to your model. For details, see Section 6.1.4: Hooking DEFINE_EXECUTE_AT_EXIT UDFs.

2.2.5 DEFINE_EXECUTE_FROM_GUI

Description

DEFINE_EXECUTE_FROM_GUI is a general-purpose macro that you can use to define a UDF which is to be executed from a user-defined graphical user interface (GUI). For example, a C function that is defined using DEFINE_EXECUTE_FROM_GUI can be executed whenever a button is clicked in a user-defined GUI. Custom GUI components (dialog boxes, buttons, etc.) are defined in ANSYS FLUENT using the Scheme language.

Usage

DEFINE_EXECUTE_FROM_GUI(name,libname,mode)

Argument Type	Description
symbol name	UDF name.
char *libname	name of the UDF library that has been loaded in ANSYS FLUENT
int mode	an integer passed from the Scheme program that defines the user-defined GUI.

Function returns
void

There are three arguments to DEFINE_EXECUTE_FROM_GUI: name, libname, and mode. You supply name, the name of the UDF. The variables libname and mode are passed by the ANSYS FLUENT solver to your UDF. The integer variable mode is passed from the Scheme program which defines the user-defined GUI, and represent the possible user options available from the GUI dialog box. A different C function in UDF can be called for each option. For example, the user-defined GUI dialog box may have a number of buttons. Each button may be represented by different integers, which, when clicked, will execute a corresponding C function.

> *i* DEFINE_EXECUTE_FROM_GUI UDFs must be implemented as compiled UDFs, and there can be only one function of this type in a UDF library.

Example

The following UDF, named `reset_udm`, resets all user-defined memory (UDM) values when a reset button on a user-defined GUI dialog box is clicked. The clicking of the button is represented by 0, which is passed to the UDF by the ANSYS FLUENT solver.

```
/***********************************************************
   UDF called from a user-defined GUI dialog box to reset
   all user-defined memory locations
***********************************************************/
#include "udf.h"

DEFINE_EXECUTE_FROM_GUI(reset_udm, myudflib, mode)
{
Domain *domain = Get_Domain(1); /* Get domain pointer */
Thread *t;
cell_t c;
int i;

/* Return if mode is not zero */
if (mode != 0) return;

/* Return if no User-Defined Memory is defined in ANSYS FLUENT */
        if (n_udm == 0) return;

        /* Loop over all cell threads in domain */
thread_loop_c(t, domain)
        {
        /* Loop over all cells */
                begin_c_loop(c, t)
                        {
                          /* Set all UDMs to zero */
                          for (i = 0; i < n_udm; i++)
                                  {
                                   C_UDMI(c, t, i) = 0.0;
                                  }
                        }
                end_c_loop(c, t);
        }
}
```

Hooking an Execute From GUI UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EXECUTE_FROM_GUI is compiled (Chapter 5: Compiling UDFs), the function will *not* need to be hooked to ANSYS FLUENT through any graphics dialog boxes. Instead, the function will be searched automatically by the ANSYS FLUENT solver when the execution of the UDF is requested (i.e., when a call is made from a user-defined Scheme program to execute a C function).

2.2.6 DEFINE_EXECUTE_ON_LOADING

Description

DEFINE_EXECUTE_ON_LOADING is a general-purpose macro that can be used to specify a function that executes as soon as a compiled UDF library is loaded in ANSYS FLUENT. This is useful when you want to initialize or setup UDF models when a UDF library is loaded. (Alternatively, if you save your case file when a shared library is loaded, then the UDF will execute whenever the case file is subsequently read.)

Compiled UDF libraries are loaded using either the Compiled UDFs or the UDF Library Manager dialog box (see Section 5.5: Load and Unload Libraries Using the UDF Library Manager Dialog Box). An EXECUTE_ON_LOADING UDF is the best place to reserve user-defined scalar (UDS) and user-defined memory (UDM) for a particular library (Sections 3.2.8 and 3.2.9) as well as set UDS and UDM names (Sections 3.2.8 and 3.2.9).

> *i*   DEFINE_EXECUTE_ON_LOADING UDFs can be executed only as compiled UDFs.

Usage

DEFINE_EXECUTE_ON_LOADING(name,libname)

Argument Type   **Description**
symbol name   UDF name.
char *libname   compiled UDF library name.

Function returns
void

There are two arguments to DEFINE_EXECUTE_ON_LOADING: name and libname. You supply a name for the UDF which will be used by ANSYS FLUENT when reporting that the EXECUTE_ON_LOADING UDF is being run. The libname is set by ANSYS FLUENT to be the name of the library (e.g., libudf) that you have specified (by entering a name or

keeping the default `libudf`). `libname` is passed so that you can use it in messages within your UDF.

Example 1

The following simple UDF named `report_version`, prints a message on the console that contains the version and release number of the library being loaded.

```
#include "udf.h"

static int version = 1;
static int release = 2;

DEFINE_EXECUTE_ON_LOADING(report_version, libname)
{
    Message("\nLoading %s version %d.%d\n",libname,version,release);
}
```

Example 2

The following source code contains two UDFs. The first UDF is an `EXECUTE_ON_LOADING` function that is used to reserve three UDMs (using `Reserve_User_Memory_Vars`) for a library and set unique names for the UDM locations (using `Set_User_Memory_Name`). The second UDF is an `ON_DEMAND` function that is used to set the values of the UDM locations after the solution has been initialized. The `ON_DEMAND` UDF sets the initial values of the UDM locations using `udm_offset`, which is defined in the on-loading UDF. Note that the on demand UDF must be executed *after* the solution is initialized to reset the initial values for the UDMs. See Sections 3.2.9 and 3.2.9 for more information on reserving and naming UDMs.

```
/**********************************************************************
This file contains two UDFs: an execute on loading UDF that reserves
three UDMs for libudf and renames the UDMs to enhance postprocessing,
and an on-demand UDF that sets the initial value of the UDMs.
**********************************************************************/
#include "udf.h"

#define NUM_UDM 3
static int udm_offset = UDM_UNRESERVED;

DEFINE_EXECUTE_ON_LOADING(on_loading, libname)
{
  if (udm_offset == UDM_UNRESERVED) udm_offset =
```

```
            Reserve_User_Memory_Vars(NUM_UDM);

  if (udm_offset == UDM_UNRESERVED)
    Message("\nYou need to define up to %d extra UDMs in GUI and "
               "then reload current library %s\n", NUM_UDM, libname);
  else
    {
      Message("%d UDMs have been reserved by the current "
                  "library %s\n",NUM_UDM, libname);

      Set_User_Memory_Name(udm_offset,"lib1-UDM-0");
      Set_User_Memory_Name(udm_offset+1,"lib1-UDM-1");
      Set_User_Memory_Name(udm_offset+2,"lib1-UDM-2");
    }
  Message("\nUDM Offset for Current Loaded Library = %d",udm_offset);
}


DEFINE_ON_DEMAND(set_udms)
{
  Domain *d;
  Thread *ct;
  cell_t c;
  int i;

  d=Get_Domain(1);

  if(udm_offset != UDM_UNRESERVED)
    {
      Message("Setting UDMs\n");

      for (i=0;i<NUM_UDM;i++)
{
  thread_loop_c(ct,d)
    {
      begin_c_loop(c,ct)
{
  C_UDMI(c,ct,udm_offset+i)=3.0+i/10.0;
}
      end_c_loop(c,ct)
    }
}
    }
  else
```

```
        Message("UDMs have not yet been reserved for library 1\n");
}
```

Hooking an Execute On Loading UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EXECUTE_ON_LOADING is compiled (Chapter 5: Compiling UDFs), the function will *not* need to be hooked to ANSYS FLUENT through any graphics dialog boxes. Instead, ANSYS FLUENT searches the newly-loaded library for any UDFs of the type EXECUTE_ON_LOADING, and will automatically execute them in the order they appear in the library.

2.2.7 DEFINE_EXECUTE_AFTER_CASE/DATA

Description

DEFINE_EXECUTE_AFTER_CASE and DEFINE_EXECUTE_AFTER_DATA are general-purpose macros that can be used to specify a function that executes after the case and/or data file is read in ANSYS FLUENT. This is useful because it provides access to UDF functions after the case and/or data file is read.

Compiled UDF libraries are loaded using either the Compiled UDFs or the UDF Library Manager dialog box (see Section 5.5: Load and Unload Libraries Using the UDF Library Manager Dialog Box).

> *i* DEFINE_EXECUTE_AFTER_CASE and DEFINE_EXECUTE_AFTER_DATA UDFs can be executed only as compiled UDFs.

Usage

DEFINE_EXECUTE_AFTER_CASE(name,libname) or
DEFINE_EXECUTE_AFTER_DATA(name,libname)

Argument Type	Description
symbol name	UDF name.
char *libname	compiled UDF library name.

Function returns
void

There are two arguments to DEFINE_EXECUTE_AFTER_CASE and DEFINE_EXECUTE_AFTER_DATA: name and libname. You supply a name for the UDF which will be used by ANSYS FLUENT when reporting that the EXECUTE_AFTER_CASE or EXECUTE_AFTER_DATA UDF is being run. The libname is set by ANSYS FLUENT to be the name of the library (e.g., libudf)

that you have specified (by entering a name or keeping the default `libudf`). `libname` is passed so that you can use it in messages within your UDF.

Example

The following simple UDF named `after_case` and `after_data`, prints a message to the console that contains the name of the library being loaded.

```
#include "udf.h"

DEFINE_EXECUTE_AFTER_CASE(after_case, libname)

{
Message("EXECUTE_AFTER_CASE called from $s\n", libname);

}
DEFINE_EXECUTE_AFTER_DATA(after_data, libname)

{
Message("EXECUTE_AFTER_DATA called from $s\n", libname);
}
```

Hooking an Execute After Reading Case and Data File UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EXECUTE_AFTER_CASE or DEFINE_EXECUTE_AFTER_DATA is compiled (Chapter 5: Compiling UDFs), the function will *not* need to be hooked to ANSYS FLUENT through any graphics dialog boxes. Instead, ANSYS FLUENT searches the newly-loaded library for any UDFs of the type EXECUTE_AFTER_CASE or EXECUTE_AFTER_DATA, and will automatically execute them in the order they appear in the library.

2.2.8 `DEFINE_INIT`

Description

`DEFINE_INIT` is a general-purpose macro that you can use to specify a set of initial values for your solution. `DEFINE_INIT` accomplishes the same result as patching, but does it in a different way, by means of a UDF. A `DEFINE_INIT` function is executed once per initialization and is called immediately after the default initialization is performed by the solver. Since it is called after the flow field is initialized, it is typically used to set initial values of flow quantities. For an overview of the ANSYS FLUENT solution process which shows when a `DEFINE_INIT` UDF is called, refer to Figures 1.9.1, 1.9.2, and 1.9.3.

Usage

`DEFINE_INIT(name,d)`

Argument Type	Description
`symbol name`	UDF name.
`Domain *d`	Pointer to the domain over which the initialization function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the pointer that is passed to the function by the solver is the mixture-level domain.

Function returns
`void`

There are two arguments to `DEFINE_INIT`: `name` and `d`. You supply `name`, the name of the UDF. `d` is passed from the ANSYS FLUENT solver to your UDF.

Example

The following UDF, named `my_init_func`, initializes flow field variables in a solution. It is executed once, at the beginning of the solution process. The function can be executed as an interpreted or compiled UDF in ANSYS FLUENT.

```
/************************************************************************
   UDF for initializing flow field variables
************************************************************************/

#include "udf.h"

DEFINE_INIT(my_init_func,d)
```

```
{
  cell_t c;
  Thread *t;
  real xc[ND_ND];

  /* loop over all cell threads in the domain  */
  thread_loop_c(t,d)
    {

      /* loop over all cells  */
      begin_c_loop_all(c,t)
        {
          C_CENTROID(xc,c,t);
          if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                          pow(xc[1] - 0.5,2.),
                          pow(xc[2] - 0.5,2.))) < 0.25)
            C_T(c,t) = 400.;
          else
            C_T(c,t) = 300.;
        }
      end_c_loop_all(c,t)
    }
}
```

The macro ND_SUM(a,b,c) computes the sum of the first two arguments (2D) or all three arguments (3D). It is useful for writing functions involving vector operations so that the same function can be used for 2D and 3D. For a 2D case, the third argument is ignored. See Chapter 3: Additional Macros for Writing UDFs for a description of predefined macros such as C_CENTROID) and ND_SUM.

Hooking an Initialization UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_INIT is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., my_init_func) will become visible and selectable via the User-Defined Function Hooks dialog box in ANSYS FLUENT. Note that you can hook multiple init functions to your model. See Section 6.1.5: Hooking DEFINE_INIT UDFs for details.

2.2.9 DEFINE_ON_DEMAND

Description

DEFINE_ON_DEMAND is a general-purpose macro that you can use to specify a UDF that is executed "on demand" in ANSYS FLUENT, rather than having ANSYS FLUENT call it automatically during the calculation. Your UDF will be executed immediately, after it is activated, but it is not accessible while the solver is iterating. Note that the domain pointer d is not explicitly passed as an argument to DEFINE_ON_DEMAND. Therefore, if you want to use the domain variable in your on-demand function, you will need to first retrieve it using the Get_Domain utility provided by ANSYS FLUENT (shown in the example below). See Section 3.2.6: Domain Pointer (Get_Domain) for details on Get_Domain.

Usage

DEFINE_ON_DEMAND(name)

Argument Type	Description
symbol name	UDF name.

Function returns
void

There is only one argument to DEFINE_ON_DEMAND: name. You supply name, the name of the UDF.

Example

The following UDF, named on_demand_calc, computes and prints the minimum, maximum, and average temperatures for the current data field. It then computes a temperature function

$$f(T) = \frac{T - T_{\min}}{T_{\max} - T_{\min}}$$

and stores it in user-defined memory location 0 (which is allocated as described in Section 3.2.3: Cell Macros). After you hook the on-demand UDF (as described in Section 6.1.6: Hooking DEFINE_ON_DEMAND UDFs), the field values for $f(T)$ will be available in drop-down lists in postprocessing dialog boxes in ANSYS FLUENT. You can select this field by choosing User Memory 0 in the User-Defined Memory... category. If you write a data file after executing the UDF, the user-defined memory field will be saved to the data file. This source code can be interpreted or compiled in ANSYS FLUENT.

```
/***********************************************************************
   UDF to calculate temperature field function and store in
   user-defined memory. Also print min, max, avg temperatures.
***********************************************************************/
#include "udf.h"

DEFINE_ON_DEMAND(on_demand_calc)
{
   Domain *d; /* declare domain pointer since it is not passed as an
                  argument to the DEFINE macro  */
   real tavg = 0.;
   real tmax = 0.;
   real tmin = 0.;
   real temp,volume,vol_tot;
   Thread *t;
   cell_t c;
   d = Get_Domain(1);      /* Get the domain using ANSYS FLUENT utility */

   /* Loop over all cell threads in the domain */
   thread_loop_c(t,d)
     {

     /* Compute max, min, volume-averaged temperature */

     /* Loop over all cells  */
     begin_c_loop(c,t)
       {
         volume = C_VOLUME(c,t);   /* get cell volume */
         temp = C_T(c,t);          /* get cell temperature */

         if (temp < tmin || tmin == 0.) tmin = temp;
         if (temp > tmax || tmax == 0.) tmax = temp;

         vol_tot += volume;
         tavg += temp*volume;

       }
     end_c_loop(c,t)

     tavg /= vol_tot;

     printf("\n Tmin = %g   Tmax = %g   Tavg = %g\n",tmin,tmax,tavg);
```

```
       /* Compute temperature function and store in user-defined memory*/
       /*(location index 0)                                            */

       begin_c_loop(c,t)
         {
           temp = C_T(c,t);
           C_UDMI(c,t,0) = (temp-tmin)/(tmax-tmin);
         }
       end_c_loop(c,t)


       }
   }
```

Get_Domain is a macro that retrieves the pointer to a domain. It is necessary to get the domain pointer using this macro since it is not explicitly passed as an argument to DEFINE_ON_DEMAND. The function, named on_demand_calc, does not take any explicit arguments. Within the function body, the variables that are to be used by the function are defined and initialized first. Following the variable declarations, a looping macro is used to loop over each cell thread in the domain. Within that loop another loop is used to loop over all the cells. Within the inner loop, the total volume and the minimum, maximum, and volume-averaged temperature are computed. These computed values are printed to the ANSYS FLUENT console. Then a second loop over each cell is used to compute the function $f(T)$ and store it in user-defined memory location 0. Refer to Chapter 3: Additional Macros for Writing UDFs for a description of predefined macros such as C_T and begin_c_loop.

Hooking an On-Demand UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_ON_DEMAND is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., on_demand_calc) will become visible and selectable in the Execute On Demand dialog box in ANSYS FLU-ENT. See Section 6.1.6: Hooking DEFINE_ON_DEMAND UDFs for details.

2.2.10 DEFINE_RW_FILE

Description

DEFINE_RW_FILE is a general-purpose macro that you can use to specify customized information that is to be written to a case or data file, or read from a case or data file. You can save and restore custom variables of any data type (e.g., integer, real, CXBoolean, structure) using DEFINE_RW_FILE. It is often useful to save dynamic information (e.g., number of occurrences in conditional sampling) while your solution is being calculated, which is another use of this function. Note that the read order and the write order must be the same when you use this function.

Usage

DEFINE_RW_FILE(name,fp)

Argument Type	Description
symbol name	UDF name.
FILE *fp	Pointer to the file you are reading or writing.

Function returns
void

There are two arguments to DEFINE_RW_FILE: name and fp. You supply name, the name of the UDF. fp is passed from the solver to the UDF.

> *i*   DEFINE_RW_FILE cannot be used in UDFs that are executed on Windows systems.

Example

The following C source code listing contains examples of functions that write information to a data file and read it back. These functions are concatenated into a single source file that can be interpreted or compiled in ANSYS FLUENT.

```
/*************************************************************************
   UDFs that increment a variable, write it to a data file
   and read it back in
**************************************************************************/
#include "udf.h"

int kount = 0;  /* define global variable kount */
```

```
DEFINE_ADJUST(demo_calc,d)
{
  kount++;
  printf("kount = %d\n",kount);
}
DEFINE_RW_FILE(writer,fp)
{
  printf("Writing UDF data to data file...\n");
  fprintf(fp,"%d",kount); /* write out kount to data file */
}
DEFINE_RW_FILE(reader,fp)
{
  printf("Reading UDF data from data file...\n");
  fscanf(fp,"%d",&kount); /* read kount from data file */
}
```

At the top of the listing, the integer `kount` is defined and initialized to zero. The first function (`demo_calc`) is an `ADJUST` function that increments the value of `kount` at each iteration, since the `ADJUST` function is called once per iteration. (See Section 2.2.1: DEFINE_ADJUST for more information about `ADJUST` functions.) The second function (`writer`) instructs ANSYS FLUENT to write the current value of `kount` to the data file, when the data file is saved. The third function (`reader`) instructs ANSYS FLUENT to read the value of `kount` from the data file, when the data file is read.

The functions work together as follows. If you run your calculation for, say, 10 iterations (`kount` has been incremented to a value of 10) and save the data file, then the current value of `kount` (10) will be written to your data file. If you read the data back into ANSYS FLUENT and continue the calculation, `kount` will start at a value of 10 and will be incremented at each iteration. Note that you can save as many static variables as you want, but you must be sure to read them in the same order in which they are written.

Hooking a Read/Write Case or Data File UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_RW_FILE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first `DEFINE` macro argument (e.g., `writer`) will become visible and selectable via the User-Defined Function Hooks dialog box in ANSYS FLUENT. Note that you can hook multiple read/write functions to your model. See Section 6.1.7: Hooking DEFINE_RW_FILE UDFs for details.

2.3 Model-Specific DEFINE Macros

The DEFINE macros presented in this section are used to set parameters for a particular model in ANSYS FLUENT. Table 2.3–Table 2.3.5 provides a quick reference guide to the DEFINE macros, the functions they are used to define, and the dialog boxes where they are activated in ANSYS FLUENT. Definitions of each DEFINE macro are listed in udf.h. For your convenience, they are listed in Appendix B.

- Section 2.3.1: DEFINE_CHEM_STEP

- Section 2.3.2: DEFINE_CPHI

- Section 2.3.3: DEFINE_DIFFUSIVITY

- Section 2.3.4: DEFINE_DOM_DIFFUSE_REFLECTIVITY

- Section 2.3.5: DEFINE_DOM_SOURCE

- Section 2.3.6: DEFINE_DOM_SPECULAR_REFLECTIVITY

- Section 2.3.7: DEFINE_EMISSIVITY_WEIGHTING_FACTOR

- Section 2.3.8: DEFINE_GRAY_BAND_ABS_COEFF

- Section 2.3.9: DEFINE_HEAT_FLUX

- Section 2.3.10: DEFINE_IGNITE_SOURCE

- Section 2.3.11: DEFINE_NET_REACTION_RATE

- Section 2.3.12: DEFINE_NOX_RATE

- Section 2.3.13: DEFINE_PR_RATE

- Section 2.3.14: DEFINE_PRANDTL UDFs

- Section 2.3.15: DEFINE_PROFILE

- Section 2.3.16: DEFINE_PROPERTY UDFs

- Section 2.3.17: DEFINE_SCAT_PHASE_FUNC

- Section 2.3.18: DEFINE_SOLAR_INTENSITY

- Section 2.3.19: DEFINE_SOURCE

- Section 2.3.20: DEFINE_SOX_RATE

- Section 2.3.21: DEFINE_SPECIFIC_HEAT

Table 2.3.1: Quick Reference Guide for Model-Specific DEFINE Functions

Function	DEFINE Macro	Dialog Box Activated In
mixing constant	DEFINE_CPHI	User-Defined Function Hooks
homogeneous net mass reaction rate for all species, integrated over a time step	DEFINE_CHEM_STEP	User-Defined Function Hooks
species mass or UDS diffusivity	DEFINE_DIFFUSIVITY	Create/Edit Materials
diffusive reflectivity for discrete ordinates (DO) model	DEFINE_DOM_DIFFUSE_ REFLECTIVITY	User-Defined Function Hooks
source for DO model	DEFINE_DOM_SOURCE	User-Defined Function Hooks
specular reflectivity for DO model	DEFINE_DOM_SPECULAR_ REFLECTIVITY	User-Defined Function Hooks
emissivity weighting factor for the RTE of the non-gray DO model	DEFINE_EMISSIVITY_ WEIGHTING_FACTOR	User-Defined Function Hooks
gray band absorption coefficient for DO model	DEFINE_GRAY_BAND_ ABS_COEFF	Create/Edit Materials
wall heat flux	DEFINE_HEAT_FLUX	User-Defined Function Hooks
ignition time source	DEFINE_IGNITE_SOURCE	User-Defined Function Hooks
homogeneous net mass reaction rate for all species	DEFINE_NET_ REACTION_RATE	User-Defined Function Hooks

Table 2.3.2: Quick Reference Guide for Model-Specific DEFINE Functions–
Continued

Function	DEFINE Macro	Dialog Box Activated In
NO$_x$ formation rates (for Thermal NO$_x$, Prompt NO$_x$, Fuel NO$_x$, and N$_2$O Pathways) and upper limit for temperature PDF	DEFINE_NOX_RATE	NOx Model
particle surface reaction rate	DEFINE_PR_RATE	User-Defined Function Hooks
Prandtl numbers	DEFINE_PRANDTL	Viscous Model
species mass fraction	DEFINE_PROFILE	boundary condition (e.g., Velocity Inlet)
velocity at a boundary	DEFINE_PROFILE	boundary condition
pressure at a boundary	DEFINE_PROFILE	boundary condition
temperature at a boundary	DEFINE_PROFILE	boundary condition
mass flux at a boundary	DEFINE_PROFILE	boundary condition
target mass flow rate for pressure outlet	DEFINE_PROFILE	Pressure Outlet
turbulence kinetic energy	DEFINE_PROFILE	boundary condition (e.g., Velocity Inlet)
turbulence dissipation rate	DEFINE_PROFILE	boundary condition
specific dissipation rate	DEFINE_PROFILE	boundary condition
porosity	DEFINE_PROFILE	boundary condition
viscous resistance	DEFINE_PROFILE	boundary condition
inertial resistance	DEFINE_PROFILE	boundary condition
porous resistance direction vector	DEFINE_PROFILE	boundary condition
user-defined scalar boundary value	DEFINE_PROFILE	boundary condition
internal emissivity	DEFINE_PROFILE	boundary condition

Table 2.3.3: Quick Reference Guide for Model-Specific DEFINE Functions–
Continued

Function	DEFINE Macro	Dialog Box Activated In
wall thermal conditions (heat flux, heat generation rate, temperature, heat transfer coefficient, external emissivity, external radiation and free stream temperature)	DEFINE_PROFILE	boundary condition
wall radiation (internal emissivity, irradiation)	DEFINE_PROFILE	boundary condition
wall momentum (shear stress x,y,z components swirl component, moving wall velocity components, roughness height, roughness constant)	DEFINE_PROFILE	boundary condition
wall species mass fractions	DEFINE_PROFILE	boundary condition
wall user-defined scalar boundary value	DEFINE_PROFILE	boundary condition
wall discrete phase boundary value	DEFINE_PROFILE	boundary condition
density (as function of temperature)	DEFINE_PROPERTY	Create/Edit Materials
density (as function of pressure for compressible liquids)	DEFINE_PROPERTY	Create/Edit Materials
viscosity	DEFINE_PROPERTY	Create/Edit Materials
mass diffusivity	DEFINE_PROPERTY	Create/Edit Materials
thermal conductivity	DEFINE_PROPERTY	Create/Edit Materials
thermal diffusion coefficient	DEFINE_PROPERTY	Create/Edit Materials

Table 2.3.4: Quick Reference Guide for Model-Specific DEFINE Functions–
Continued

Function	DEFINE Macro	Dialog Box Activated In
absorption coefficient	DEFINE_PROPERTY	Create/Edit Materials
scattering coefficient	DEFINE_PROPERTY	Create/Edit Materials
laminar flow speed	DEFINE_PROPERTY	Create/Edit Materials
rate of strain	DEFINE_PROPERTY	Create/Edit Materials
speed of sound function	DEFINE_PROPERTY	Create/Edit Materials
user-defined mixing law for mixture materials (density viscosity, thermal conductivity)	DEFINE_PROPERTY	Create/Edit Materials
scattering phase function	DEFINE_SCAT_PHASE_FUNC	Create/Edit Materials
solar intensity	DEFINE_SOLAR_INTENSITY	Radiation Model
mass source	DEFINE_SOURCE	cell zone condition
momentum source	DEFINE_SOURCE	cell zone condition
energy source	DEFINE_SOURCE	cell zone condition
turbulence kinetic energy source	DEFINE_SOURCE	cell zone condition
turbulence dissipation rate source	DEFINE_SOURCE	cell zone condition
species mass fraction source	DEFINE_SOURCE	cell zone condition
user-defined scalar source	DEFINE_SOURCE	cell zone condition
P1 radiation model source	DEFINE_SOURCE	cell zone condition
SO_x formation rate and upper limit for temperature PDF	DEFINE_SOX_RATE	SOx Model
specific heat and sensible enthalpy	DEFINE_SPECIFIC_HEAT	Create/Edit Materials
surface reaction rate	DEFINE_SR_RATE	User-Defined Function Hooks
transition correlation numbers	DEFINE_TRANS	Viscous Model
turbulent premixed source	DEFINE_TURB_PREMIX_SOURCE	User-Defined Function Hooks
turbulent Schmidt number	DEFINE_TURB_SCHMIDT	Viscous Model
turbulent viscosity	DEFINE_TURBULENT_VISCOSITY	Viscous Model
volume reaction rate	DEFINE_VR_RATE	User-Defined Function Hooks
wall function	DEFINE_WALL_FUNCTIONS	Viscous Model

Table 2.3.5: Quick Reference Guide for Model-Specific DEFINE Functions
MULTIPHASE ONLY

Function	DEFINE Macro	Dialog Box Activated In
volume fraction (all multiphase models)	DEFINE_PROFILE	boundary condition
contact angle (VOF)	DEFINE_PROFILE	Wall boundary condition
heat transfer coefficient (Eulerian)	DEFINE_PROPERTY	Phase Interaction
surface tension coefficient (VOF)	DEFINE_PROPERTY	Phase Interaction
cavitation surface tension coefficient (Mixture)	DEFINE_PROPERTY	Phase Interaction
cavitation vaporization pressure (Mixture)	DEFINE_PROPERTY	Phase Interaction
particle or droplet diameter (Mixture)	DEFINE_PROPERTY	Create/Edit Materials
diameter (Eulerian, Mixture)	DEFINE_PROPERTY	Secondary Phase
solids pressure (Eulerian, Mixture)	DEFINE_PROPERTY	Secondary Phase
radial distribution (Eulerian, Mixture)	DEFINE_PROPERTY	Secondary Phase
elasticity modulus (Eulerian, Mixture)	DEFINE_PROPERTY	Secondary Phase
viscosity (Eulerian, Mixture)	DEFINE_PROPERTY	Secondary Phase
temperature (Eulerian, Mixture)	DEFINE_PROPERTY	Secondary Phase
bulk viscosity (Eulerian)	DEFINE_PROPERTY	Secondary Phase
frictional viscosity (Eulerian)	DEFINE_PROPERTY	Secondary Phase
frictional pressure (Eulerian)	DEFINE_PROPERTY	Secondary Phase
frictional modulus (Eulerian)	DEFINE_PROPERTY	Secondary Phase
granular viscosity (Eulerian)	DEFINE_PROPERTY	Secondary Phase
granular bulk viscosity (Eulerian)	DEFINE_PROPERTY	Secondary Phase
granular conductivity (Eulerian)	DEFINE_PROPERTY	Secondary Phase
temperature source (Eulerian, Mixture)	DEFINE_SOURCE	boundary condition

2.3.1 DEFINE_CHEM_STEP

Description

You can use DEFINE_CHEM_STEP to specify the change in mass fraction due to homogeneous reaction over a time step:

$$Y_i^{\Delta t} = Y_i^0 + \int_0^{\Delta t} \frac{dY_i}{dt} dt \qquad (2.3\text{-}1)$$

where Y_i^0 is the initial mass fraction of species i, t is time, Δt is the given time step, and $\frac{dY_i}{dt}$ is the net rate of change of the ith species mass fraction. $Y_i^{\Delta t}$ is ith species mass fraction at the end of the integration.

DEFINE_CHEM_STEP UDFs are used for the Laminar finite-rate (with Stiff Chemistry enabled), EDC and PDF Transport models.

Usage

DEFINE_CHEM_STEP(name,c,t,p,num_p,n_spe,dt,pres,temp,yk)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index of current particle.
Thread *t	Pointer to cell thread for particle.
Particle *p	Pointer to particle data structure that contains data related to the particle being tracked.
int num_p	Not Used.
int n_spec	Number of volumetric species.
double *dt	Time step.
double *pres	Pointer to pressure.
double *temp	Pointer to temperature.
double *yk	Pointer to array of initial species mass fractions.

Function returns
void

There are nine arguments to DEFINE_CHEM_STEP: name, c, p ,num_p, n_spe, dt, pres, temp, and yk. You supply name, the name of the UDF. c, p, n_spe, dt, pres, temp, and yk are variables that are passed by the ANSYS FLUENT solver to your UDF. num_p is not used by the function and can be ignored. The output of the function is the array of mass fractions yk *after* the integration step. The initial mass fractions in array yk are overwritten.

Example

The following UDF, named `user_chem_step`, assumes that the net volumetric reaction rate is the expression,

$$\frac{dY_k}{dt} = 1/N_{spe} - Y_k \tag{2.3-2}$$

where N_{spe} is the number of species.

An analytic solution exists for the integral of this ODE as,

$$Y_k^{\Delta t} = (Y_k^0 - 1/N_{spe})exp(-\Delta t) + 1/N_{spe} \tag{2.3-3}$$

```
/****************************************************
   Example UDF that demonstrates DEFINE_CHEM_STEP
****************************************************/
#include "udf.h"

DEFINE_CHEM_STEP(user_chem_step,cell,thread,particle,nump,nspe,dt,pres,temp,yk)
{
   int i;
   double c = 1./(double)nspe;
   double decay = exp(-(*dt));
   for(i=0;i<n_spe;i++)
     yk[i] = (yk[i]-c)*decay + c;
}
```

Hooking a Chemistry Step UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_CHEM_STEP` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., `user_chem_step`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.1: Hooking `DEFINE_CHEM_STEP` UDFs for details.

2.3.2 DEFINE_CPHI

Description

You can use DEFINE_CPHI to set the value of the mixing constant C_ϕ (see Equation 11.3-4 and Equation 11.3-6 in the separate Theory Guide for details). It is useful for modeling flows where C_ϕ departs substantially from its default value of 2, which occurs at low Reynolds and/or high Schmidt numbers.

Usage

DEFINE_CPHI(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_CPHI: name, c, and t. You supply name, the name of the UDF. c and t are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the real value of the mixing constant (C_ϕ) and return it to the solver.

Hooking a Mixing Constant UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_CPHI is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT whenever the Composition PDF Transport model is enabled. See Section 6.2.2: Hooking DEFINE_CPHI UDFs for details.

2.3.3 DEFINE_DIFFUSIVITY

Description

You can use DEFINE_DIFFUSIVITY to specify the diffusivity for the species transport equations (e.g., mass diffusivity) or for user-defined scalar (UDS) transport equations. For details about UDS diffusivity, see Section 8.6: User-Defined Scalar (UDS) Diffusivity in the separate User's Guide.

Usage

DEFINE_DIFFUSIVITY(name,c,t,i)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the diffusivity function is to be applied.
int i	Index that identifies the species or user-defined scalar.

Function returns
real

There are four arguments to DEFINE_DIFFUSIVITY: name, c, and t, and i. You supply name, the name of the UDF. c, t, and i are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the diffusivity *only* for a single cell and return the real value to the solver.

Note that diffusivity UDFs are called by ANSYS FLUENT from within a loop on cell threads. Consequently, your UDF will not need to loop over cells in a thread since ANSYS FLUENT is doing it outside of the function call.

Example

The following UDF, named `mean_age_diff`, computes the diffusivity for the mean age of air using a user-defined scalar. Note that the mean age of air calculations do not require that energy, radiation, or species transport calculations have been performed. You will need to set `uds-0 = 0.0` at all inlets and outlets in your model. This function can be executed as an interpreted or compiled UDF.

```
/*************************************************************************
   UDF that computes diffusivity for mean age using a user-defined
   scalar.
*************************************************************************/

#include "udf.h"

DEFINE_DIFFUSIVITY(mean_age_diff,c,t,i)
{
    return C_R(c,t) * 2.88e-05 + C_MU_EFF(c,t) / 0.7;
}
```

Hooking a Diffusivity UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_DIFFUSIVITY` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name that you specified in the `DEFINE` macro argument (e.g., `mean_age_diff`) will become visible and selectable in the **Create/Edit Materials** dialog box in ANSYS FLUENT. See Section 6.2.3: Hooking `DEFINE_DIFFUSIVITY` UDFs for details.

2.3.4 DEFINE_DOM_DIFFUSE_REFLECTIVITY

Description

You can use DEFINE_DOM_DIFFUSE_REFLECTIVITY to modify the inter-facial reflectivity computed by ANSYS FLUENT at diffusely reflecting semi-transparent walls, based on the refractive index values. During execution, a DEFINE_DOM_DIFFUSE_REFLECTIVITY function is called by ANSYS FLUENT for each semi-transparent wall and also for each band (in the case of a non-gray discrete ordinates (DO) model). Therefore the function can be used to modify diffuse reflectivity and diffuse transmissivity values at the interface.

Usage

DEFINE_DOM_DIFFUSE_REFLECTIVITY(name,t,nb,n_a,n_b,diff_ref_a,diff_tran_a,
diff_ref_b,diff_tran_b)

> **i** Note that all of the arguments to a DEFINE macro need to be placed on the same line in your source code. Splitting the DEFINE statement onto several lines will result in a compilation error.

Argument Type	Description
symbol name	UDF name.
Thread *t	Pointer to the thread on which the discrete ordinate diffusivity function is to be applied.
int nb	Band number (needed for the non-gray discrete ordinates (DO) model).
real n_a	Refractive index of medium a.
real n_b	Refractive index of medium b.
real *diff_ref_a	Diffuse reflectivity at the interface facing medium a.
real *diff_tran_a	Diffuse transmissivity at the interface facing medium a.
real *diff_ref_b	Diffuse reflectivity at the interface facing medium b.
real *diff_tran_b	Diffuse transmissivity at the interface facing medium b.

Function returns
void

There are nine arguments to DEFINE_DOM_DIFFUSE_REFLECTIVITY: name, t, nb, n_a, n_b, diff_ref_a, diff_tran_a, diff_ref_b, and diff_tran_b. You supply name, the name of the UDF. t, nb, n_a, n_b, diff_ref_a, diff_tran_a, diff_ref_b, and diff_tran_b are variables that are passed by the ANSYS FLUENT solver to your UDF.

Example

The following UDF, named `user_dom_diff_refl`, modifies diffuse reflectivity and transmissivity values on both the sides of the interface separating medium a and b. The UDF is called for all the semi-transparent walls and prints the value of the diffuse reflectivity and transmissivity values for side a and b.

> *i* Note that in the example that follows, the `DEFINE_DOM_DIFFUSE_REFLECTIVITY` statement is broken up into two lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/* UDF to print the diffuse reflectivity and transmissivity
at semi-transparent walls*/

#include "udf.h"

DEFINE_DOM_DIFFUSE_REFLECTIVITY(user_dom_diff_refl,t,nband,n_a,n_b,
diff_ref_a,diff_tran_a,diff_ref_b,diff_tran_b)
{
printf("diff_ref_a=%f  diff_tran_a=%f \n", *diff_ref_a, *diff_tran_a);
printf("diff_ref_b=%f  diff_tran_b=%f \n", *diff_ref_b, *diff_tran_b);
}
```

Hooking a Discrete Ordinates Model (DOM) Diffuse Reflectivity UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_DOM_DIFFUSE_REFLECTIVITY` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first `DEFINE` macro argument (e.g., `user_dom_diff_refl`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT.

See Section 6.2.4: Hooking `DEFINE_DOM_DIFFUSE_REFLECTIVITY` UDFs for details.

2.3.5 DEFINE_DOM_SOURCE

Description

You can use DEFINE_DOM_SOURCE to modify the emission term (first term on the right hand side in Equation 5.3-37 or Equation 5.3-38 in the separate Theory Guide as well as the scattering term (second term on the right hand side of either equation) in the radiative transport equation for the discrete ordinates (DO) model.

Usage

DEFINE_DOM_SOURCE(name,c,t,ni,nb,emission,in_scattering,abs_coeff,scat_coeff)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.
int ni	Direction represented by the solid angle.
int nb	Band number (needed for the non-gray discrete ordinates (DO) model).
real *emission	Pointer to emission term in the radiative transport equation (Equation 5.3-37 in the separate Theory Guide).
real *in_scattering	Pointer to scattering term in the radiative transport equation (Equation 5.3-38 in the separate Theory Guide).
real *abs_coeff	Pointer to absorption coefficient.
real *scat_coeff	Pointer to scattering coefficient.

Function returns
void

There are nine arguments to DEFINE_DOM_SOURCE: name, c, ni, nb, emission, in_scattering, abs_coeff, and scat_coeff. You supply name, the name of the UDF. c, ni, nb, emission, in_scattering, abs_coeff, and scat_coeff are variables that are passed by the ANSYS FLUENT solver to your UDF. DEFINE_DOM_SOURCE is called by ANSYS FLUENT for each cell.

Example

In the following UDF, named `dom`, the emission term present in the radiative transport equation is modified. The UDF is called for all the cells and increases the emission term by 5%.

```
/* UDF to alter the emission source term in the DO model */

#include "udf.h"

DEFINE_DOM_SOURCE(dom,c,t,ni,nb,emission,in_scattering,abs_coeff,scat_coeff)
{

    /* increased the emission by 5 %   */

    *emission  *= 1.05;

}
```

Hooking a DOM Source UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_DOM_SOURCE` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first `DEFINE` macro argument (e.g., `dom`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. Note that you can hook multiple discrete ordinate source term functions to your model. See Section 6.2.5: Hooking `DEFINE_DOM_SOURCE` UDFs for details.

2.3.6 `DEFINE_DOM_SPECULAR_REFLECTIVITY`

Description

You can use `DEFINE_DOM_SPECULAR_REFLECTIVITY` to modify the inter-facial reflectivity of specularly reflecting semi-transparent walls. You may wish to do this if the reflectivity is dependent on other conditions that the standard boundary condition doesn't allow for (see Section 5.3.6: Specular Semi-Transparent Walls in the separate Theory Guide for more information). During ANSYS FLUENT execution, the same UDF is called for all the faces of the semi-transparent wall, for each of the directions.

Usage

```
DEFINE_DOM_SPECULAR_REFLECTIVITY(name,f,t,nband,n_a,n_b,ray_direction,en,
internal_reflection,specular_reflectivity,specular_transmissivity)
```

> *i* Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Face index.
`Thread *t`	Pointer to face thread on which the specular reflectivity function is to be applied.
`int nband`	Band number (needed for non-gray discrete ordinates (DO) model).
`real n_a`	Refractive index of medium a.
`real n_b`	Refractive index of medium b.
`real ray_direction`	Direction vector (s) defined in Equation 5.3-55 in the separate Theory Guide
`real en`	Interface normal vector (n) defined in Equation 5.3-55 in the separate Theory Guide
`int internal_reflection`	Variable used to flag the code that total internal reflection has occurred.
`real *specular_reflectivity`	Specular reflectivity for the given direction s.
`real *specular_transmissivity`	Specular transmissivity for the given direction s.

Function returns
`void`

There are eleven arguments to `DEFINE_DOM_SPECULAR_REFLECTIVITY`: `name`, `f`, `t`, `nband`, `n_a`, `n_b`, `ray_direction`, `en`, `internal_reflection`, `specular_reflectivity`, and `specular_transmissivity`. You supply `name`, the name of the UDF. `f`, `t`, `nband`, `n_a`, `n_b`, `ray_direction`, `en`, `internal_reflection`, `specular_reflectivity`, and `specular_transmissivity` are variables that are passed by the ANSYS FLUENT solver to your UDF.

Example

In the following UDF, named `user_dom_spec_refl`, specular reflectivity and transmissivity values are altered for a given ray direction s at face `f`.

> *i* Note that in the example that follows, the `DEFINE_DOM_SPECULAR_REFLECTIVITY` statement is broken up into three lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/*  UDF to alter the specular reflectivity and transmissivity, at
    semi-transparent walls, along direction s at face f */

#include "udf.h"

DEFINE_DOM_SPECULAR_REFLECTIVITY(user_dom_spec_refl,f,t, nband,n_a,n_b,
ray_direction,en,internal_reflection,specular_reflectivity,
specular_transmissivity)
{
   real angle, cos_theta;
   real PI = 3.141592;
   cos_theta = NV_DOT(ray_direction, en);
   angle = acos(cos_theta);
   if (angle >45   && angle < 60)
     {
       *specular_reflectivity = 0.3;
       *specular_transmissivity = 0.7;
     }
}
```

Hooking a Discrete Ordinates Model (DOM) Specular Reflectivity UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DOM_SPECULAR_REFLECTIVITY is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_dom_spec_refl) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT.

See Section 6.2.6: Hooking DEFINE_DOM_SPECULAR_REFLECTIVITY UDFs for details.

2.3.7   DEFINE_EMISSIVITY_WEIGHTING_FACTOR

Description

When employing the non-gray discrete ordinates (DO) radiation model, you can use DEFINE_EMISSIVITY_WEIGHTING_FACTOR to modify the emissivity weighting factor $F(0 \to n\lambda_2 T) - F(0 \to n\lambda_1 T)$. By default, the emissivity weighting factor is calculated internally by ANSYS FLUENT so it can be used in the emission term of the radiative transfer equation, as shown in Equation 5.3-39 in the separate Theory Guide. This macro allows you to revise the calculated value.

Usage

```
DEFINE_EMISSIVITY_WEIGHTING_FACTOR(name,c,t,T,nb,emissivity_weighting_factor)
```

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.
real T	Temperature.
int nb	Band number
real *emissivity_weighting_factor	The emissivity weighting factor in the emission term of the RTE for the non-gray DO model (Equation 5.3-39 in the separate Theory Guide).

Function returns
```
void
```

There are six arguments to DEFINE_EMISSIVITY_WEIGHTING_FACTOR: name, c, t, T, nb, emissivity_weighting_factor. You supply name, the name of the UDF. c, t, T, nb, and emissivity_weighting_factor are variables that are passed by the ANSYS FLU-ENT solver to your UDF. DEFINE_EMISSIVITY_WEIGHTING_FACTOR is called by ANSYS FLUENT for each cell.

Example

In the following UDF (named em_wt), the emissivity weighting factor present in the emission term of the RTE for the non-gray DO model is modified. The UDF is called for all of the cells. It modifies the emissivity weighting factor so that it is no longer the value calculated internally by ANSYS FLUENT, but is instead changed to 1.

```
/* UDF to alter the emissivity weighting factor for the non-gray DO model */

#include "udf.h"

DEFINE_EMISSIVITY_WEIGHTING_FACTOR(em_wt,c,t,T,nb,emissivity_weighting_factor)
{

   /* revise the calculated emissivity_weighting_factor to be a value of 1  */

   *emissivity_weighting_factor = 1.0;

}
```

Hooking a Emissivity Weighting Factor UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EMISSIVITY_WEIGHTING_FACTOR is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., em_wt) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.7: Hooking DEFINE_EMISSIVITY_WEIGHTING_FACTOR UDFs for details.

2.3.8 DEFINE_GRAY_BAND_ABS_COEFF

Description

You can use DEFINE_GRAY_BAND_ABS_COEFF to specify a UDF for the gray band absorption coefficient as a function of temperature, that can be used with a non-gray discrete ordinates model.

Usage

DEFINE_GRAY_BAND_ABS_COEFF(name,c,t,nb)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.
int nb	Band number associated with non-gray model.

Function returns
real

There are four arguments to DEFINE_GRAY_BAND_ABS_COEFF: name, c, t, and nb. You supply name, the name of the UDF. The variables c, t, and nb are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value of the gray band coefficient to the solver.

Example

The following UDF, named user_gray_band_abs, specifies the gray-band absorption coefficient as a function of temperature that can be used for a non-gray discrete ordinates model.

```
#include   "udf.h"

DEFINE_GRAY_BAND_ABS_COEFF(user_gray_band_abs,c,t,nb)
{
   real abs_coeff = 0;
   real T = C_T(c,t);

  switch (nb)
  {
     case 0 :  abs_coeff = 1.3+0.001*T; break;
     case 1 :  abs_coeff =  2.7 + 0.005*T; break;
}

return abs_coeff;

}
```

Hooking a Gray Band Coefficient UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_GRAY_BAND_ABS_COEFF is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_gray_band_abs) will become visible and selectable in the Create/Edit Materials dialog box for the Absorption Coefficient.

See Section 6.2.8: Hooking DEFINE_GRAY_BAND_ABS_COEFF UDFs for details.

2.3.9 DEFINE_HEAT_FLUX

Description

You can use DEFINE_HEAT_FLUX to modify the heat flux at a wall. Despite the name, a DEFINE_HEAT_FLUX UDF is *not* the means to specify the actual heat flux entering a domain from the outside. To specify this type of heat flux, you would simply use a DEFINE_PROFILE function in conjunction with a heat flux thermal boundary condition. In contrast, a DEFINE_HEAT_FLUX UDF allows you to modify the way in which the dependence between the flux entering the domain and the wall and cell temperatures is modeled.

i This function allows you to modify the heat flux at walls adjacent to a solid.Note, however, that for solids since only heat conduction is occurring, any extra heat flux that you add in a heat flux UDF can have a detrimental effect on the solution of the energy equation. These effects will likely show up in conjugate heat transfer problems. To avoid this, you will need to make sure that your heat flux UDF excludes the walls adjacent to solids, or includes only the necessary walls adjacent to fluid zones.

Usage

DEFINE_HEAT_FLUX(name,f,t,c0,t0,cid,cir)

Argument Type	Description
symbol name	UDF name.
face_t f	Index that identifies a wall face.
Thread *t	Pointer to wall face thread on which heat flux function is to be applied.
cell_t c0	Cell index that identifies the cell next to the wall.
Thread *t0	Pointer to the adjacent cell's thread.
real cid[]	Array of fluid-side diffusive heat transfer coefficients.
real cir[]	Array of radiative heat transfer coefficients.

Function returns
void

There are seven arguments to DEFINE_HEAT_FLUX: name, f, t, c0, t0, cid, and cir. You supply name, the name of the UDF. f, t, c0, and t0 are variables that are passed by the ANSYS FLUENT solver to your UDF. Arrays cir[] and cid[] contain the linearizations of the radiative and diffusive heat fluxes, respectively, computed by ANSYS FLUENT based on the activated models. These arrays allow you to modify the heat flux in any

way that you choose. ANSYS FLUENT computes the heat flux at the wall using these arrays *after* the call to DEFINE_HEAT_FLUX, so the total heat flux at the wall will be the currently computed heat flux (based on the activated models) with any modifications as defined by your UDF.

The diffusive heat flux (`qid`) and radiative heat flux (`qir`) are computed by ANSYS FLUENT according to the following equations:

```
qid = cid[0] + cid[1]*C_T(c0,t0) - cid[2]*F_T(f,t) - cid[3]*pow(F_T(f,t),4)
qir = cir[0] + cir[1]*C_T(c0,t0) - cir[2]*F_T(f,t) - cir[3]*pow(F_T(f,t),4)
```

The sum of `qid` and `qir` defines the total heat flux from the fluid to the wall (this direction being positive flux), and, from an energy balance at the wall, equals the heat flux of the surroundings (exterior to the domain). Note that heat flux UDFs (defined using DEFINE_HEAT_FLUX) are called by ANSYS FLUENT from within a loop over wall faces.

i In order for the solver to compute C_T and F_T, the values you supply to `cid[1]` and `cid[2]` should never be zero.

Example

Section 8.2.5: Implementing ANSYS FLUENT's P-1 Radiation Model Using User-Defined Scalars provides an example of the P-1 radiation model implementation through a user-defined scalar. An example of the usage of the DEFINE_HEAT_FLUX macro is included in that implementation.

Hooking a Heat Flux UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_HEAT_FLUX is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., `heat_flux`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.9: Hooking DEFINE_HEAT_FLUX UDFs for details.

2.3.10 DEFINE_IGNITE_SOURCE

Description

You can use DEFINE_IGNITE_SOURCE to customize the ignition time source term in the autoignition model model.

Usage

DEFINE_IGNITE_SOURCE(name,c,t,source)

Argument Type	Description
symbol name	UDF name
cell_t c	Cell index
Thread *t	Pointer to cell thread on which the ignition source term is to be applied
real *source	Pointer to the ignition source term

Function returns
void

There are four arguments to DEFINE_IGNITE_SOURCE: name, c, t, and source. You supply name, the name of the UDF. c, t and source are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the value referenced by the source pointer as shown in the example below.

Example

The following UDF, named ign_udf_src, specifies a custom source term in the ignition model. The source code must be executed as a compiled UDF in ANSYS FLUENT.

In the standard ignition model in ANSYS FLUENT, the source term for the ignition progress variable is given by a Livengood-Wu integral [7]:

$$S_{ig} = \int_{t=t_0}^{t} \frac{dt}{\tau_{ig}} \tag{2.3-4}$$

where dt is the flow time step and τ_{ig} is the correlation between spark time and knock, by Douaud and Eyzat [3]:

$$\tau = 0.01768 \left(\frac{ON}{100}\right)^{3.402} p^{-1.7} \exp\left(\frac{3800}{T}\right) \tag{2.3-5}$$

Here, ON is the octane number of the fuel, p is the absolute pressure in atmospheres and T is the temperature in Kelvin.

In the following UDF example, the Douaud and Eyzat correlation is used to calculate an induction time. Please see Chapter 3: Additional Macros for Writing UDFs for details on the `NNULLP`, `C_STORAGE_R`, `C_PREMIXC_T`, `C_P`, `C_R`, `CURRENT_TIMESTEP` and `C_IGNITE` macros used below.

```
/*------------------------------------------------------------------*/
/* This UDF produces an ignition model source in ANSYS FLUENT 12.0  */
/* that uses the default parameters for the correlation of Douaud   */
/* and Eyzat for knock.                                             */
/*------------------------------------------------------------------*/

#include "udf.h"

real A = 0.01768;              /* Preexponential                    */
real Ea = 3800;                /* Activation temperature            */
real O_N = 90.0;               /* Octane number                     */
real O_E = 3.402;              /* Octane number exponent            */
real P_E = -1.7;               /* Pressure exponent                 */

static real A1 = 0.0;          /* Cached value of A*ON^OE           */
static real dt = 0.0;          /* Cached time step                  */
static real p_op = 0.0;        /* Cached value of operating pressure */
static cxboolean lit = FALSE;  /* Cached burning flag               */

DEFINE_IGNITE_SOURCE(ign_udf_src, c, t, source)
{
  real rho = C_R(c,t);
  real time = 0.0;
  real prog = NNULLP(THREAD_STORAGE(t,SV_PREMIXC_M1))  ?
    C_STORAGE_R(c,t,SV_PREMIXC_M1) :
    C_STORAGE_R(c,t,SV_PREMIXC) ;
  real fuel = 1.0 - prog;
  real T = C_PREMIXC_T(c,t);
  real P = C_P(c,t);
  real ipv = C_IGNITE(c,t);

  if (c == 0)
    {
      dt = CURRENT_TIMESTEP;
      p_op = RP_Get_Real("operating-pressure");
      A1 = A * pow(O_N/100,O_E);
    }
```

```
  if (ipv > 1.0)
    lit = TRUE;

  P += p_op;
  P /= 101325.;        /* in atm */
  P = MAX(P,0.01);    /* minimum pressure for ignition */

  if (fuel > 0.99 || lit)
    time = A1 * pow(P,P_E) * exp(Ea/T);

  if (time > 0.0)
    {
      real max_source = rho*(5.0-ipv)/dt;
      real user_source = rho/time;
      *source = MIN(user_source,max_source);
    }
  else
    *source = 0.0;

  return;
}
```

Hooking an Ignition Source UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_IGNITE_SOURCE is compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., ign_udf_src) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.10: Hooking DEFINE_IGNITE_SOURCE UDFs for details.

2.3.11 DEFINE_NET_REACTION_RATE

Description

You can use DEFINE_NET_REACTION_RATE to compute the homogeneous net molar reaction rates of all species. The net reaction rate of a species is the sum over all reactions of the volumetric reaction rates:

$$R_i = \sum_{r=1}^{N_R} \hat{R}_{i,r} \tag{2.3-6}$$

where R_i is the net reaction rate of species i and $\hat{R}_{i,r}$ is the Arrhenius molar rate of creation/destruction of species i in reaction r.

A DEFINE_NET_REACTION_RATE UDF may be used for the laminar finite-rate (with stiff chemistry enabled), EDC, and PDF Transport models, as well as for the surface chemistry model. In contrast, the volumetric UDF function DEFINE_VR_RATE and surface UDF function DEFINE_SR_RATE are used for the laminar finite-rate model when stiff chemistry is disabled.

Usage

DEFINE_NET_REACTION_RATE(name,c,t,particle,pressure,temp,yi,rr,jac)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index of current particle.
Thread *t	Pointer to cell thread for particle.
Particle *particle	Pointer to Particle data structure that contains data related to the particle being tracked.
double *pressure	Pointer to pressure variable.
double *temp	Pointer to temperature variable.
double *yi	Pointer to array containing species mass fractions.
double *rr	Pointer to array containing net molar reaction rates.
double *jac	Pointer to array of Jacobians.

Function returns
void

There are nine arguments to DEFINE_NET_REACTION_RATE: name, c, t, particle, pressure, temp, yi, rr, and jac. You supply name, the name of the UDF. The variables c, t, particle, pressure, temp, yi, rr, and jac are passed by the ANSYS FLUENT solver to your UDF and have SI units. The outputs of the function are the array of net molar

reaction rates, `rr` (with units $kmol/m^3 - s$), and the Jacobian array `jac`. The Jacobian is only required for surface chemistry, and is the derivative of the surface net reaction rate with respect to the species concentration.

DEFINE_NET_REACTION_RATE is called for all fluid zones (volumetric reactions as well as surface reactions in porous media) and for all wall thread zones whenever the Reaction option is enabled in the boundary conditions dialog box and the UDF is hooked to ANSYS FLUENT in the User-Defined Function Hooks dialog box.

> **i** DEFINE_NET_REACTION_RATE functions can be executed only as compiled UDFs.

Example

The following UDF, named `net_rxn`, assumes that the net volumetric reaction rate is the expression,

$$R_{net} = 1/N_{spe} - Y_i \qquad (2.3\text{-}7)$$

where N_{spe} is the number of species.

```
/***********************************************************
    Net Reaction Rate Example UDF
***********************************************************/
#include "udf.h"

DEFINE_NET_REACTION_RATE(net_rxn,c,t,particle,pressure,temp,yi,rr,jac)
{
    int i;
    for(i=0;i<n_spe;i++)
      rr[i] = 1./(real)n_spe - yi[i];
}
```

Note that during the coarse of the ODE solution, the species mass fractions can exceed realizable bounds. For optimal ODE performance, the species mass fractions should not be clipped, but derived quantities, such as concentrations which are raised to non-integer powers, must be bounded. Also, if density is required, for instance to calculate concentrations, it should be calculated from the temperature and species passed into the UDF. Finally, double precision should be used for all local variables.

Hooking a Net Reaction Rate UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_NET_REACTION_RATE` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., `net_rxn`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.11: Hooking `DEFINE_NET_REACTION_RATE` UDFs for details.

2.3.12 `DEFINE_NOX_RATE`

Description

You can use the `DEFINE_NOX_RATE` to specify a custom NO_x rate for thermal NO_x, prompt NO_x, fuel NO_x, and N_2O intermediate pathways that can *either* replace the internally-calculated NO_x rate in the source term equation, or be added to the ANSYS FLUENT rate. Example 1 demonstrates this use of `DEFINE_NOX_RATE`. By default, the Add to FLUENT Rate option is enabled UDF Rate group box in each of the tabs under Formation Model Parameters, so that user-defined rates are added to the ANSYS FLUENT-calculated rates. You can change this default by selecting Replace FLUENT Rate, so that the ANSYS FLUENT-calculated rate for that NO_x pathway will not be used and it will instead be replaced by the NO_x rate you have defined in your UDF.

> **_i_** Note that a single UDF is used to define the different rates for the four NO_x pathways: thermal NO_x, prompt NO_x, fuel NO_x, and N_2O intermediate pathway. That is, a NO_x rate UDF can contain up to four separate rate functions that are concatenated in a single source file which you hook to ANSYS FLUENT.

`DEFINE_NOX_RATE` may also be used to calculate the upper limit for the integration of the temperature PDF (when temperature is accounted for in the turbulence interaction modeling). You can calculate a custom maximum limit (T_{max}) for each cell and then assign it to the `POLLUT_CTMAX(Pollut_Par)` macro (see Section 3.2.7: NO_x Macros for further details about data access macros). Example 2 demonstrates this use of `DEFINE_NOX_RATE`.

> **_i_** If you want to use `DEFINE_NOX_RATE` only for the purpose of specifying T_{max}, then be sure that the user-defined NO_x rate does not alter the internally-calculated rate for the source term calculation.

Usage

```
DEFINE_NOX_RATE(name,c,t,Pollut,Pollut_Par,NOx)
```

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the NO_x rate is to be applied.
Pollut_Cell *Pollut	Pointer to the data structure that contains the common data at each cell
Pollut_Parameter *Pollut_Par	Pointer to the data structure that contains auxiliary data.
NOx_Parameter *NOx	Pointer to the data structure that contains data specific to the NO_x model.

Function returns
```
void
```

There are six arguments to DEFINE_NOX_RATE: name, c, t, Pollut, Pollut_Par, and NOx. You will supply name, the name of the UDF. c, t, Pollut, Pollut_Par, and NOx are variables that are passed by the ANSYS FLUENT solver to your function. A DEFINE_NOX_RATE function does not output a value. The calculated NO_x rates (or other pollutant species rates) are returned through the Pollut structure as the forward rate POLLUT_FRATE(Pollut) and reverse rate POLLUT_RRATE(Pollut), respectively.

i The data contained within the NO_x structure is specific *only* to the NO_x model. Alternatively, the Pollut structure contains data at each cell that are useful for all pollutant species (e.g., forward and reverse rates, gas phase temperature, density). The Pollut_Par structure contains auxiliary data common to all pollutant species (e.g., equation solved, universal gas constant, species molecular weights). Note that molecular weights extracted from the Pollut_Par structure (i.e., Pollut_Par->sp[IDX(i)].mw for pollutant species—NO, HCN, etc.—and Pollut_Par->sp[i].mw for other species, such as O_2) has units of kg/kg-mol.

Example 1

The following compiled UDF, named user nox, exactly reproduces the default ANSYS FLUENT NO_x rates for the prompt NO_x pathway. Note that this UDF will replace the ANSYS FLUENT rate *only* if you select Replace FLUENT Rate in the UDF Rate group box in the Prompt tab. Otherwise, the rate computed in the UDF will be added to ANSYS FLUENT's default rate. See Section 6.2.12: Hooking DEFINE NOX RATE UDFs for details.

See Section 3.2.7: NO_x Macros for details about the NO_x macros (e.g., POLLUT EQN, MOLECON, ARRH) that are used in pollutant rate calculations in this UDF.

```
/*********************************************************************
   UDF example of User-Defined NOx Rate for ANSYS FLUENT 12 or later

   If used with the "Replace with UDF" radio buttons activated,
   this UDF will exactly reproduce the default ANSYS FLUENT NOx
   rates for prompt NOx pathway.

   The flag "Pollut_Par->pollut_io_pdf == IN_PDF" should always
   be used for rates other than that from char N, so that if
   requested, the contributions will be PDF integrated. Any
   contribution from char must be included within a switch
   statement of the form "Pollut_Par->pollut_io_pdf == OUT_PDF".
   *
   * Arguments:
   *    char nox_func_name          - UDF name
   *    cell_t c                    - Cell index
   *    Thread *t                   - Pointer to cell thread on
   *                                  which the NOx rate is to be
   *                                  applied
   *    Pollut_Cell *Pollut         - Pointer to Pollut structure
   *    Pollut_Parameter *Pollut_Par - Pointer to Pollut_Par
   *                                  structure
   *    NOx_Parameter *NOx          - Pointer to NOx structure

   *******************************************************************/

#include "udf.h"

DEFINE_NOX_RATE(user_nox, c, t, Pollut, Pollut_Par, NOx)
{
/* NOx->prompt_nox = Flag to indicate Prompt NOx is enabled
 * NOx->prompt_udf_replace = Flag to indicate UDF replace
 * Pollut_Par->nfstreams = Number of fuel streams
```

```
* Pollut_Par->nfspe[i] = Number of fuel species in stream "i"
* NOx->equiv_ratio[i] = Equivalence ratio for stream "i"
* NOx->c_number[i] = Carbon number for stream "i"
* Pollut_Par->fuel_idx[j][i] = Index of jth species in stream "i"
* Pollut_Par->fuel_dup[j][i] = Fuel species duplication check
* Pollut_Par->uni_R = Universal gas constant in SI units
* Pollut->temp_m = Mean gas temperature (K)
* Pollut->press = Pressure in SI units
* Pollut->oxy_order = Oxygen order (please refer to user manual)
*/
 POLLUT_FRATE(Pollut) = 0.0;
 POLLUT_RRATE(Pollut) = 0.0;

 switch (Pollut_Par->pollut_io_pdf) {
 case IN_PDF:
   /* Included source terms other than those from char */

   if (POLLUT_EQN(Pollut_Par) == EQ_NO) {

     /* Prompt NOx */
     if (NOx->prompt_nox && NOx->prompt_udf_replace) {
       int ifstream;
       real f=0., rf;

       Rate_Const K_PM = {6.4e6,  0.0, 36483.49436};

       for(ifstream=0; ifstream<Pollut_Par->nfstreams; ifstream++) {
         int i;
         real xc_fuel=0., eqr=NOx->equiv_ratio[ifstream];
         for (i=0; i<Pollut_Par->nfspe[ifstream]; i++) {
           if(!Pollut_Par->fuel_dup[i][ifstream])
             xc_fuel += MOLECON(Pollut, Pollut_Par->fuel_idx[i][ifstream]);
         }
         f += (4.75 + 0.0819*NOx->c_number[ifstream]
           - 23.2*eqr + 32.0*pow(eqr, 2.) - 12.2*pow(eqr, 3.))*xc_fuel;
       }
       rf = ARRH(Pollut, K_PM);
       rf *= pow((Pollut_Par->uni_R*Pollut->temp_m/Pollut->press),
                 (1.+Pollut->oxy_order));
       rf *= pow(MOLECON(Pollut, O2), Pollut->oxy_order);
       rf *= MOLECON(Pollut, N2);

       POLLUT_FRATE(Pollut) += f*rf;
```

```
      }

    }
    break;

  case OUT_PDF:
    /* Char Contributions, must be included here */
    break;

  default:
    /* Not used */
    break;
  }
}
```

Example 2

The following compiled UDF, named `nox_func_name`, specifies a custom maximum limit
(T_{max}) for the integration of the temperature PDF for each cell. Note that this UDF
does not alter the internally-calculated NO_x rate.

See Section 3.2.7: NO_x Macros for details about the NO_x macro (`POLLUT_CTMAX`) used in
this UDF.

```
/**************************************************************
   UDF example of User-Defined Tmax value

   *
   * Arguments:
   *    char nox_func_name          - UDF name
   *    cell_t c                    - Cell index
   *    Thread *t                   - Pointer to cell thread
   *                                  on which the NOx rate
   *                                  is to be applied
   *    Pollut_Cell *Pollut         - Pointer to Pollut_Cell
   *                                  structure
   *    Pollut_Parameter *Pollut_Par - Pointer to Pollut_Parameter
   *                                  structure
   *    NOx_Parameter *NOx          - Pointer to NOx_Parameter
   *                                  structure

   ANSYS FLUENT Version: 12.0 or later
**************************************************************/
```

```
#include "udf.h"

int ud_nox_do_once=1;

enum
{
  CELL_TMAX=0,
  N_REQUIRED_UDM
};

/*Compute/assign Tmax at each cell*/
real ud_eval_cell_tmax(cell_t c,Thread *t)
{
  real tmax = 0.;

  /* Compute cell-based Tmax value */
  tmax = 1.1*C_T(c,t); /* This is only an example */

  return tmax;
}

DEFINE_NOX_RATE(user_nox, c, t, Pollut, Pollut_Par, NOx)
{
  /* Assign cell-based Tmax value */
  POLLUT_CTMAX(Pollut_Par) = ud_eval_cell_tmax(c,t);
  /*POLLUT_CTMAX(Pollut_Par) = C_UDMI(c,t,CELL_TMAX);*/
}

DEFINE_ON_DEMAND(init_tmax)
{
  Domain *domain;
  register Thread *t;
  register cell_t c;

  Message("Computing/Storing cell Tmax values\n");
  domain = Get_Domain(1);

  /* Store User-Defined Tmax at each cell */
  if(ud_nox_do_once == 1) {
    if(n_udm < N_REQUIRED_UDM)
      Error("Not enough udm allocated\n");

    thread_loop_c (t,domain)
```

```
      begin_c_loop (c,t)
        C_UDMI(c,t,CELL_TMAX) = ud_eval_cell_tmax(c,t);
      end_c_loop (c,t)
    ud_nox_do_once = 0;
  }
  Message("Computing cell Tmax values completed..\n");
}
```

Hooking a NO$_x$ Rate UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_NOX_RATE is compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_nox) will become visible and selectable in the NOx Model dialog box in ANSYS FLUENT. See Section 6.2.12: Hooking DEFINE_NOX_RATE UDFs for details.

2.3.13 DEFINE_PR_RATE

Description

You can use DEFINE_PR_RATE to specify a custom particle surface reaction for the multiple surface reactions particle model. During ANSYS FLUENT execution, the same UDF is called sequentially for all particle surface reactions, so DEFINE_PR_RATE can be used to define custom reaction rates for a single reaction, or for multiple reactions. The volumetric and wall surface reactions are not affected by the definition of this macro and will follow the designated rates. Note that a DEFINE_PR_RATE UDF is *not* called with the coupled solution option, so you will need to disable the Coupled Heat Mass Solution option in the Discrete Phase Model dialog box when using it. The auxiliary function, zbrent_pr_rate, which is provided below, can be used when there is no analytical solution for the overall particle reaction rate.

Usage

DEFINE_PR_RATE(name,c,t,r,mw,ci,p,sf,dif_index,cat_index,rr)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index of current particle.
Thread *t	Pointer to cell thread for particle.
Reaction *r	Pointer to data structure that represents the current reaction.
real *mw	Pointer to array containing gaseous and surface species molecular weights
real *ci	Pointer to array containing gas partial pressures.
Tracked_Particle *p	Pointer to Tracked_Particle data structure that contains data related to the particle being tracked.
real *sf	Pointer to array containing mass fractions of the solid species in the particle char mass at the current time step.
int dif_index	Diffusion controlled species as defined in the Reactions dialog box for the current reaction.
int cat_index	Catalyst species as defined in the Reactions dialog box for the current reaction.
real *rr	Pointer to array containing particle reaction rate (kg/s).

Function returns
void

There are eleven arguments to DEFINE_PR_RATE: name, c, t, r, mw, ci, p, sf, dif_index, cat_index, and rr. You supply name, the name of the UDF. c, t, r, mw, ci, p, sf, dif_index, cat_index, and rr are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the value referenced by the real pointer rr to the particle reaction rate in kg/s.

Note that p is an argument to many particle-specific macros defined in Section 3.2.7: DPM Macros and can be used to obtain information about particle properties. Also note that the order in which the solid species mass fractions are stored in array sf is the same as the order in which the species are defined in the Selected Solid Species list in the Create/Edit Materials dialog box, which is opened from the Edit Species names option for the Mixture Material.

DEFINE_PR_RATE is called by ANSYS FLUENT every time step during the particle tracking calculation. The auxiliary function zbrent_pr_rate is used when there is no analytical solution for the overall particle reaction rate. It uses Brent's method to find the root of a function known to lie between $x1$ and $x2$. The root will be refined until its accuracy has reached tolerance tol. This is demonstrated in Example 2.

Auxiliary function

```
zbrent_pr_rate(real (*func),(real,real [],int [],cxboolean [],char *,) real
ruser[],int iuser[],
cxboolean buser[],char *cuser,real x1 real x2,real tol,cxboolean *ifail)
```

Auxiliary function returns: real

Example 1

The following UDF, named user_pr_rate, specifies a particle reaction rate given by Equation 7.3-9 in the separate Theory Guide, where the effectiveness factor η_r is defined as

$$\eta_r = 1 - x$$

where x is the fractional conversion of the particle char mass. In this case, the UDF will be applied to all surface particle reactions defined in the ANSYS FLUENT model.

```
/* UDF of specifying the surface reaction rate of a particle */

#include "udf.h"

#define A1  0.002
#define E1  7.9e7

DEFINE_PR_RATE(user_pr_rate,c,t,r,mw,pp,p,sf,dif_i,cat_i,rr)
{
/* Argument types
    cell_t c
    Thread *t
    Reaction *r (reaction structure)
    real *mw  (species molecular weight)
    real *pp  (gas partial pressures)
    Tracked_Particle *p (particle structure)
    real *sf   (current mass fractions of solid species in
                particle char mass)
```

```
    int dif_i  (index of diffusion controlled species)
    int cat_i  (index of catalyst species)
    real *rr   (rate of reaction kg/s)
*/

real ash_mass =
P_INIT_MASS(p)*(1.-DPM_CHAR_FRACTION(p)-DPM_VOLATILE_FRACTION(p));

real one_minus_conv =
MAX(0.,(P_MASS(p) -ash_mass) / P_INIT_MASS(p)/ DPM_CHAR_FRACTION(p));

real rate = A1*exp(-E1/UNIVERSAL_GAS_CONSTANT/P_T(p));

*rr=-rate*P_DIAM(p)*P_DIAM(p)*M_PI*sf[0]*one_minus_conv;
}
```

Example 2

The following compiled UDF, named `user_rate`, specifies a particle reaction rate given
by Equation 7.3-4 to Equation 7.3-7 in the separate Theory Guide. The reaction order
on the kinetic rate is 0.9 and the effectiveness factor η_r is defined as

$$\eta_r = 1 - x$$

where x is the fractional conversion of the particle char mass. In this case it is necessary
to obtain a numerical solution for the overall surface reaction rate.

This UDF is called only for reaction 2, which means that the default ANSYS FLUENT
solution will be used for the rest of the particle surface reactions defined.

```
/* UDF of specifying the surface reaction rate of a particle,
 using a numerical solution */

#include "udf.h"

#define c1  5e-12
#define A1  0.002
#define E1  7.9e7
#define tolerance 1e-4
#define order 0.9

real reaction_rate(real rate, real ruser[], int iuser[], cxboolean buser[],
 char *cuser)
```

```
/* Note that all arguments in the reaction_rate function
call in your .c source file MUST be on the same line or a
compilation error will occur */


{
    return (ruser[2]*pow(MAX(0.,(ruser[0]-rate/ruser[1])),order) -rate);
}

DEFINE_PR_RATE(user_rate,c,t,r,mw,pp,p,sf,dif_i,cat_i,rr)
{
if (!strcmp(r->name, "reaction-2"))
  {
  cxboolean ifail=FALSE;

  real ash_mass =
  P_INIT_MASS(p)*(1.-DPM_CHAR_FRACTION(p)-DPM_VOLATILE_FRACTION(p));

  real one_minus_conv =
  MAX(0.,(P_MASS(p) -ash_mass) / P_INIT_MASS(p)/ DPM_CHAR_FRACTION(p));

  real ruser[3];
  int iuser[1];
  cxboolean buser[1];
  char cuser[30];

  real ratemin, ratemax, root;

  ruser[0] = pp[dif_i];
  ruser[1] = MAX(1.E-15, (c1*pow(0.5*(P_T(p)+C_T(c,t)),0.75)/P_DIAM(p)));
  ruser[2] = A1*exp(-E1/UNIVERSAL_GAS_CONSTANT/P_T(p));
  strcpy(cuser, "reaction-2");

  ratemin=0;
  ratemax=ruser[1]*pp[dif_i];

  /* arguments for auxiliary function zbrent_pr_rate */

  root = zbrent_pr_rate(reaction_rate, ruser, iuser, buser, cuser,
                        ratemin, ratemax, tolerance, &ifail);

  if (ifail) root=MAX(1.E-15,ruser[1]);
```

```
   *rr=-root*P_DIAM(p)*P_DIAM(p)*M_PI*sf[0]*one_minus_conv;

   Message("Fail status %d\n", ifail);
   Message("Reaction rate for reaction %s : %g\n", cuser, *rr);

 }
}
```

In this example, a real function named `reaction_rate` is defined at the top of the UDF. The arguments of `reaction_rate` are `real rate`, and the pointer arrays `real ruser[]`, `integer iuser[]`, `cxboolean buser[]`, and `char *cuser`, which must be declared and defined in the main body of the `DEFINE_PR_RATE` function.

Typically, if the particle surface reaction rate is described by

```
        rate = f(ruser[],iuser[],rate)
```

then the real function (in this example `reaction_rate`) should return

```
        f(ruser[],iuser[],rate) - rate
```

The variables `cxboolean buser[]` and `char *cuser` can be used to control the flow of the program in cases of complicated rate definitions.

`ratemin` and `ratemax`, hold the minimum and maximum possible values of the variable rate, respectively. They define the search interval where the numerical algorithm will search for the root of the equation, as defined in the function `reaction_rate`. The value of reaction rate `rr` will be refined until an accuracy specified by the value of tolerance `tol` is reached.

The variable `ifail` will take the value `TRUE` if the root of the function has not been found.

Hooking a Particle Reaction Rate UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_PR_RATE` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., `user_pr_rate`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.13: Hooking `DEFINE_PR_RATE` UDFs for details.

2.3.14 DEFINE_PRANDTL **UDFs**

The following DEFINE macros can be used to specify Prandtl numbers in ANSYS FLUENT, for single-phase flows.

DEFINE_PRANDTL_D

Description

You can use DEFINE_PRANDTL_D to specify Prandtl numbers for turbulent dissipation (ϵ).

Usage

DEFINE_PRANDTL_D(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index of cell on which the Prandtl number function is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_PRANDTL_D: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value for the turbulent dissipation Prandtl number to the solver.

Example

An example of a Prandtl_D UDF is provided below in the source listing for DEFINE_PRANDTL_K.

Hooking a Prandtl Number UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_D is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_pr_d) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.14: Hooking DEFINE_PRANDTL UDFs for details.

DEFINE_PRANDTL_K

Description

You can use DEFINE_PRANDTL_K to specify Prandtl numbers for turbulence kinetic energy (k).

Usage

DEFINE_PRANDTL_K(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the Prandtl number function is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_PRANDTL_K: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value for the kinetic energy Prandtl number to the solver.

Example

The following UDF implements a high-Re version of the RNG model, using the k-ϵ option that is activated in ANSYS FLUENT.

Three steps are required:

1. Set Cmu, C1eps, and C2eps as in the RNG model.

2. Calculate Prandtl numbers for k and ϵ using the UDF.

3. Add the -r source term in the ϵ equation.

In the RNG model, diffusion in k and ϵ equations appears as

$$(\mu + \mu_t) * \alpha$$

while in the standard k-ϵ model, it is given by

$$\mu + \frac{\mu_t}{Pr}$$

For the new implementation, a UDF is needed to define a Prandtl number Pr as

$$Pr = \frac{\mu_t}{[(\mu + \mu_t) * \alpha - \mu]}$$

in order to achieve the same implementation as the original RNG Model.

The following functions (which are concatenated into a single C source code file) demonstrate this usage. Note that the source code must be executed as a compiled UDF.

```
#include "udf.h"

DEFINE_PRANDTL_K(user_pr_k,c,t)
{
  real pr_k, alpha;
  real mu    = C_MU_L(c,t);
  real mu_t  = C_MU_T(c,t);

  alpha = rng_alpha(1., mu + mu_t, mu);

  pr_k = mu_t/((mu+mu_t)*alpha-mu);

  return pr_k;
}

DEFINE_PRANDTL_D(user_pr_d,c,t)
{
  real pr_d, alpha;
  real mu    = C_MU_L(c,t);
  real mu_t  = C_MU_T(c,t);

  alpha = rng_alpha(1., mu + mu_t, mu);

  pr_d = mu_t/((mu+mu_t)*alpha-mu);

  return pr_d;
}

DEFINE_SOURCE(eps_r_source,c,t,dS,eqn)
```

```
{
  real con, source;
  real mu    = C_MU_L(c,t);
  real mu_t  = C_MU_T(c,t);
  real k     = C_K(c,t);
  real d     = C_D(c,t);
  real prod  = C_PRODUCTION(c,t);

  real s =  sqrt(prod/(mu+ mu_t) ) ;
  real eta   = s*k/d;
  real eta_0 = 4.38;
  real term = mu_t*s*s*s/(1.0 + 0.012*eta*eta*eta);

  source = - term * (1. - eta/eta_0);
  dS[eqn] = - term/d;

  return source;
}
```

Hooking a Prandtl Number UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_K is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_pr_k) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.14: Hooking DEFINE_PRANDTL UDFs for details.

DEFINE_PRANDTL_O

Description

You can use DEFINE_PRANDTL_O to specify Prandtl numbers for specific dissipation (ω in the k-ω model).

Usage

DEFINE_PRANDTL_O(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the Prandtl number function is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_PRANDTL_O: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value for the specific dissipation Prandtl number to the solver.

Example

```
/* Specifying a Constant Specific Dissipation Prandtl Number */
#include "udf.h"

DEFINE_PRANDTL_O(user_pr_o,c,t)
{
  real pr_o;
  pr_o = 2.;
  return pr_o;
}
```

Hooking a Prandtl Number UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_O is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_pr_o) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.14: Hooking DEFINE_PRANDTL UDFs for details.

DEFINE_PRANDTL_T

Description

You can use DEFINE_PRANDTL_T to specify Prandtl numbers that appear in the temperature equation diffusion term.

Usage

DEFINE_PRANDTL_T(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the Prandtl number function is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_PRANDTL_T: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value for the temperature Prandtl number to the solver.

Example

```
/* Specifying a Constant Temperature Prandtl Number */
#include "udf.h"

DEFINE_PRANDTL_T(user_pr_t,c,t)
{
  real pr_t;
  pr_t = 0.85;
  return pr_t;
}
```

Hooking a Prandtl Number UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_T is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_pr_t) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.14: Hooking DEFINE_PRANDTL UDFs for details.

DEFINE_PRANDTL_T_WALL

Description

You can use DEFINE_PRANDTL_T_WALL to specify Prandtl numbers for thermal wall functions.

Usage

DEFINE_PRANDTL_T_WALL(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the Prandtl number function is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_PRANDTL_T_WALL: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the `real` value for the thermal wall function Prandtl number to the solver.

Example

```
/*****************************************************************
  Specifying a constant thermal wall function Prandtl number
  ******************************************************** **/
#include "udf.h"

DEFINE_PRANDTL_T_WALL(user_pr_t_wall,c,t)
{
  real pr_t_wall;
  pr_t_wall = 0.85;
  return pr_t_wall;
}
```

Hooking a Prandtl Number UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_T_WALL is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_pr_t_wall) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.14: Hooking DEFINE_PRANDTL UDFs for details.

2.3.15 DEFINE_PROFILE

Description

You can use DEFINE_PROFILE to define a custom boundary profile that varies as a function of spatial coordinates or time. Some of the variables you can customize at a boundary are:

- velocity, pressure, temperature, turbulence kinetic energy, turbulence dissipation rate

- mass flux

- target mass flow rate as a function of physical flow time

- species mass fraction (species transport)

- volume fraction (multiphase models)

- wall thermal conditions (temperature, heat flux, heat generation rate, heat transfer coefficients, and external emissivity, etc.)

- wall roughness conditions

- wall shear and stress conditions

- porosity

- porous resistance direction vector

- wall adhesion contact angle (VOF multiphase model)

Note that DEFINE_PROFILE allows you to modify only a single value for wall heat flux. Single values are used in the explicit source term which ANSYS FLUENT does not linearize. If you want to linearize your source term for wall heat flux and account for conductive and radiative heat transfer separately, you will need to use DEFINE_HEAT_FLUX to specify your UDF.

Some examples of boundary profile UDFs are provided below. For an overview of the ANSYS FLUENT solution process which shows when a DEFINE_PROFILE UDF is called, refer to Figures 1.9.1, 1.9.2, and 1.9.3.

Usage

```
DEFINE_PROFILE(name,t,i)
```

Argument Type	Description
symbol name	UDF name.
Thread *t	Pointer to thread on which boundary condition is to be applied.
int i	Index that identifies the variable that is to be defined. i is set when you hook the UDF with a variable in a boundary conditions dialog box through the graphical user interface. This index is subsequently passed to your UDF by the ANSYS FLUENT solver so that your function knows which variable to operate on.

Function returns
void

There are three arguments to DEFINE_PROFILE: name, t, and i. You supply name, the name of the UDF. t and i are variables that are passed by the ANSYS FLUENT solver to your UDF.

While DEFINE_PROFILE is usually used to specify a profile condition on a boundary face zone, it can also be used to specify, or fix, flow variables that are held constant during computation in a cell zone. See Section 7.2.4: Fixing the Values of Variables in the separate User's Guide for more information on fixing values in a cell zone boundary condition. For these cases, the arguments of the macro will change accordingly.

Note that unlike source term and property UDFs, profile UDFs (defined using DEFINE_PROFILE) are *not* called by ANSYS FLUENT from within a loop on threads in the boundary zone. The solver passes only the pointer to the thread associated with the boundary zone to the DEFINE_PROFILE macro. Your UDF will need to do the work of looping over all of the faces in the thread, computing the face value for the boundary variable, and then storing the value in memory. ANSYS FLUENT has provided you with a face looping macro to loop over all faces in a thread (begin_f_loop...). See Chapter 3: Additional Macros for Writing UDFs for details.

F_PROFILE is typically used along with DEFINE_PROFILE and is a predefined macro supplied by ANSYS FLUENT. F_PROFILE stores a boundary condition in memory for a given face and thread and is nested within the face loop as shown in the examples below. It is important to note that the index i that is an argument to DEFINE_PROFILE is the same argument to F_PROFILE. F_PROFILE uses the thread pointer t, face identifier f, and index i to set the appropriate boundary face value in memory. See Section 3.2.6: Set Boundary Condition Value (F_PROFILE) for a description of F_PROFILE. Note that in the case of porosity profiles, you can also utilize C_PROFILE to define those types of functions. See

the example UDFs provided below.

In multiphase cases a `DEFINE_PROFILE` UDF may be called more than once (particularly if the profile is used in a mixture domain thread). If this needs to be avoided, then add the prefix `MP_` to the UDF name. The function will then be called only once even if it is used for more than one profile.

Example 1 - Pressure Profile

The following UDF, named `pressure_profile`, generates a parabolic pressure profile according to the equation

$$p(y) = 1.1 \times 10^5 - 0.1 \times 10^5 \left(\frac{y}{0.0745} \right)^2$$

Note that this UDF assumes that the mesh is generated such that the origin is at the geometric center of the boundary zone to which the UDF is to be applied. y is 0.0 at the center of the inlet and extends to ± 0.0745 at the top and bottom of the inlet. The source code can be interpreted or compiled in ANSYS FLUENT.

```
/************************************************************************
   UDF for specifying steady-state parabolic pressure profile boundary
   profile for a turbine vane
 ************************************************************************/

#include "udf.h"

DEFINE_PROFILE(pressure_profile,t,i)
{
  real x[ND_ND];                /* this will hold the position vector */
  real y;
  face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      y = x[1];
      F_PROFILE(f,t,i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
    }
  end_f_loop(f,t)
}
```

The function named `pressure_profile` has two arguments: `t` and `i`. `t` is a pointer to the face's thread, and `i` is an integer that is a numerical label for the variable being set within each loop.

Within the function body variable f is declared as a face. A one-dimensional array x and variable y are declared as `real` data types. Following the variable declarations, a looping macro is used to loop over each face in the zone to create a profile, or an array of data. Within each loop, F_CENTROID returns the value of the face centroid (array x) for the face with index f that is on the thread pointed to by t. The y coordinate stored in x[1] is assigned to variable y, and is then used to calculate the pressure. This value is then assigned to F_PROFILE which uses the integer i (passed to it by the solver, based on your selection of the UDF as the boundary condition for pressure in the Pressure Inlet dialog box) to set the pressure face value in memory.

Example 2 - Velocity, Turbulent Kinetic Energy, and Turbulent Dissipation Rate Profiles

In the following example, DEFINE_PROFILE is used to generate profiles for the x velocity, turbulent kinetic energy, and dissipation rate, respectively, for a 2D fully-developed duct flow. Three separate UDFs named x_velocity, k_profile, and dissip_profile are defined. These functions are concatenated in a single C source file and can be interpreted or compiled in ANSYS FLUENT.

The 1/7th power law is used to specify the x velocity component:

$$v_x = v_{x,\text{free}} \left(\frac{y}{\delta} \right)^{1/7}$$

A fully-developed profile occurs when δ is one-half the duct height. In this example, the mean x velocity is prescribed and the peak (free-stream) velocity is determined by averaging across the channel.

The turbulent kinetic energy is assumed to vary linearly from a near-wall value of

$$k_{\text{nw}} = \frac{u_\tau^2}{\sqrt{C_\mu}}$$

to a free-stream value of

$$k_{\text{inf}} = 0.002 u_{\text{free}}^2$$

The dissipation rate is given by

$$\epsilon = \frac{C_\mu^{3/4}(k^{3/2})}{\ell}$$

where the mixing length ℓ is the minimum of κy and 0.085δ. (κ is the von Karman constant $= 0.41$.)

The friction velocity and wall shear take the forms:

$$u_\tau = \sqrt{\tau_w/\rho}$$

$$\tau_w = \frac{f\rho u_{\text{free}}^2}{2}$$

The friction factor is estimated from the Blasius equation:

$$f = 0.045 \left(\frac{u_{\text{free}}\delta}{\nu}\right)^{-1/4}$$

```
/**************************************************************************
      Concatenated UDFs for fully-developed turbulent inlet profiles
**************************************************************************/

#include "udf.h"

#define YMIN 0.0                              /* constants   */
#define YMAX 0.4064
#define UMEAN 1.0
#define B 1./7.
#define DELOVRH 0.5
#define VISC 1.7894e-05
#define CMU 0.09
#define VKC 0.41


/*  profile for x-velocity    */


DEFINE_PROFILE(x_velocity,t,i)
{
  real y, del, h, x[ND_ND], ufree;      /* variable declarations */
  face_t f;

  h = YMAX - YMIN;
  del = DELOVRH*h;
  ufree = UMEAN*(B+1.);

  begin_f_loop(f,t)
```

```
     {
        F_CENTROID(x,f,t);
        y = x[1];

        if (y <= del)
           F_PROFILE(f,t,i) = ufree*pow(y/del,B);
        else
           F_PROFILE(f,t,i) = ufree*pow((h-y)/del,B);
     }
  end_f_loop(f,t)
}

/*  profile for kinetic energy  */

DEFINE_PROFILE(k_profile,t,i)
{
  real y, del, h, ufree, x[ND_ND];
  real ff, utau, knw, kinf;
  face_t f;

  h = YMAX - YMIN;
  del = DELOVRH*h;
  ufree = UMEAN*(B+1.);
  ff = 0.045/pow(ufree*del/VISC,0.25);
  utau=sqrt(ff*pow(ufree,2.)/2.0);
  knw=pow(utau,2.)/sqrt(CMU);
  kinf=0.002*pow(ufree,2.);

  begin_f_loop(f,t)
    {
       F_CENTROID(x,f,t);
       y=x[1];

       if (y <= del)
         F_PROFILE(f,t,i)=knw+y/del*(kinf-knw);
       else
         F_PROFILE(f,t,i)=knw+(h-y)/del*(kinf-knw);
    }
  end_f_loop(f,t)
}

/* profile for dissipation rate  */
```

```
DEFINE_PROFILE(dissip_profile,t,i)
{
  real y, x[ND_ND], del, h, ufree;
  real ff, utau, knw, kinf;
  real mix, kay;
  face_t f;

  h = YMAX - YMIN;
  del = DELOVRH*h;
  ufree = UMEAN*(B+1.);
  ff = 0.045/pow(ufree*del/VISC,0.25);
  utau=sqrt(ff*pow(ufree,2.)/2.0);
  knw=pow(utau,2.)/sqrt(CMU);
  kinf=0.002*pow(ufree,2.);

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      y=x[1];

      if (y <= del)
        kay=knw+y/del*(kinf-knw);
      else
        kay=knw+(h-y)/del*(kinf-knw);

      if (VKC*y < 0.085*del)
        mix = VKC*y;
      else
        mix = 0.085*del;

      F_PROFILE(f,t,i)=pow(CMU,0.75)*pow(kay,1.5)/mix;
    }
  end_f_loop(f,t)
}
```

Example 3 - Fixed Velocity UDF

In the following example DEFINE_PROFILE is used to fix flow variables that are held constant during computation in a cell zone. Three separate UDFs named fixed_u, fixed_v, and fixed_ke are defined in a single C source file. They specify fixed velocities that simulate the transient startup of an impeller in an impeller-driven mixing tank. The physical impeller is simulated by fixing the velocities and turbulence quantities using the fix option in ANSYS FLUENT. See Section 7.2.4: Fixing the Values of Variables in the separate User's Guide for more information on fixing variables.

```
/**************************************************************************
   Concatenated UDFs for simulating an impeller using fixed velocity
 **************************************************************************/

#include "udf.h"

#define FLUID_ID 1
#define ua1 -7.1357e-2
#define ua2 54.304
#define ua3 -3.1345e3
#define ua4 4.5578e4
#define ua5 -1.9664e5

#define va1 3.1131e-2
#define va2 -10.313
#define va3 9.5558e2
#define va4 -2.0051e4
#define va5 1.1856e5

#define ka1 2.2723e-2
#define ka2 6.7989
#define ka3 -424.18
#define ka4 9.4615e3
#define ka5 -7.7251e4
#define ka6 1.8410e5

#define da1 -6.5819e-2
#define da2 88.845
#define da3 -5.3731e3
#define da4 1.1643e5
#define da5 -9.1202e5
#define da6 1.9567e6
```

```
DEFINE_PROFILE(fixed_u,t,i)
{
  cell_t c;
  real x[ND_ND];
  real r;

  begin_c_loop(c,t)
    {
/* centroid is defined to specify position dependent profiles */

      C_CENTROID(x,c,t);
      r =x[1];
      F_PROFILE(c,t,i) =
ua1+(ua2*r)+(ua3*r*r)+(ua4*r*r*r)+(ua5*r*r*r*r);
}
  end_c_loop(c,t)
}


DEFINE_PROFILE(fixed_v,t,i)
{
  cell_t c;
  real x[ND_ND];
  real r;

  begin_c_loop(c,t)
    {
/* centroid is defined to specify position dependent profiles*/

      C_CENTROID(x,c,t);
      r =x[1];
      F_PROFILE(c,t,i) =
va1+(va2*r)+(va3*r*r)+(va4*r*r*r)+(va5*r*r*r*r);
}
  end_c_loop(c,t)
}

DEFINE_PROFILE(fixed_ke,t,i)
{
  cell_t c;
  real x[ND_ND];
```

```
    real r;

    begin_c_loop(c,t)
      {
/* centroid is defined to specify position dependent profiles*/
        C_CENTROID(x,c,t);
        r =x[1];
        F_PROFILE(c,t,i) =
ka1+(ka2*r)+(ka3*r*r)+(ka4*r*r*r)+(ka5*r*r*r*r)+(ka6*r*r*r*r*r);
      }
    end_c_loop(c,t)
}
```

Example 4 - Wall Heat Generation Rate Profile

The following UDF, named `wallheatgenerate`, generates a heat generation rate profile
for a planar conduction wall. After it has been interpreted or compiled, you can activate
this UDF in the Wall boundary conditions dialog box in ANSYS FLUENT.

```
/*  Wall Heat Generation Rate Profile UDF  */

#include "udf.h"

DEFINE_PROFILE(wallheatgenerate,thread,i)
{
  real source = 0.001;
  face_t f;

  begin_f_loop(f,thread)
    F_PROFILE(f,thread,i) = source;
  end_f_loop(f,thread)
}
```

Example 5 - Beam Direction Profile at Semi-Transparent Walls

The following UDF, named q_nx, where x is the direction vector i, j, k, specifies the beam
direction normal to every face on the cylinder. After it has been interpreted or compiled,
you can activate this UDF in the Wall boundary conditions dialog box in ANSYS FLUENT.

```
/*  Beam Direction Profile UDF at Semi-Transparent Walls  */

#include "udf.h"

DEFINE_PROFILE(q_ni, t, position)
{
    real A[3], e_n[3];
    face_t f;
    real At;
    begin_f_loop(f, t)
      {
        F_AREA(A, f, t);
        At = NV_MAG(A);
        NV_VS(e_n,=,A,/,At);
        F_PROFILE(f, t, position) = -e_n[0];
      }
    end_f_loop(f, t)

}


DEFINE_PROFILE(q_nj, t, position)
{
    real A[3], e_n[3];
    face_t f;
    real At;
    begin_f_loop(f, t)
      {
        F_AREA(A, f, t);
        At = NV_MAG(A);
        NV_VS(e_n,=,A,/,At);
        F_PROFILE(f, t, position) = -e_n[1];
      }
    end_f_loop(f, t)

}
```

```
DEFINE_PROFILE(q_nk, t, position)
{
    real A[3], e_n[3];
    face_t f;
    real At;
    begin_f_loop(f, t)
      {
        F_AREA(A, f, t);
        At = NV_MAG(A);
        NV_VS(e_n,=,A,/,At);
        F_PROFILE(f, t, position) = -e_n[2];
      }
    end_f_loop(f, t)

}
```

Example 6 - Viscous Resistance Profile in a Porous Zone

You can either use F PROFILE or C PROFILE to define a viscous resistance profile in a porous zone. Below are two sample UDFs that demonstrate the usage of F PROFILE and C PROFILE, respectively. Note that porosity functions are hooked to ANSYS FLUENT in the Porous Zone tab in the appropriate Fluid cell zone conditions dialog box.

The following UDF, named vis res, generates a viscous resistance profile in a porous zone. After it has been interpreted or compiled and loaded, you can activate this UDF in the Fluid cell zone condition dialog box in ANSYS FLUENT.

```
/* Viscous Resistance Profile UDF in a Porous Zone  that utilizes F_PROFILE*/

#include "udf.h"

DEFINE_PROFILE(vis_res,t,i)
{
  real x[ND_ND];
  real a;
  cell_t c;

  begin_c_loop(c,t)
  {
    C_CENTROID(x,c,t);
    if( x[1] < (x[0]-0.01) )
      a = 1e9;
```

```
      else
         a = 1.0;
      F_PROFILE(c,t,i) = a;
   }
   end_c_loop(c,t)
}


/*  Viscous Resistance Profile UDF in a Porous Zone that utilizes C_PROFILE*/

#include "udf.h"

DEFINE_PROFILE(porosity_function, t, nv)
{
   cell_t c;
   begin_c_loop(c,t)
      C_PROFILE(c,t,nv) = USER INPUT  ;
   end_c_loop(c,t)
}
```

Example 7 - Porous Resistance Direction Vector

The following UDF contains profile functions for two porous resistance direction vectors that utilize C_PROFILE. These profiles can be hooked to corresponding direction vectors under Porous Zone in the Fluid cell zone condition dialog box.

```
/*  Porous Resistance Direction Vector Profile that utilizes C_PROFILE*/

#include "udf.h"

DEFINE_PROFILE{dir1, t, nv}
{
   cell_t c;
   begin_c_loop(c,t)
      C_PROFILE(c,t,nv) = USER INPUT1  ;
   end_c_loop(c,t)
}


DEFINE_PROFILE{dir2, t, nv}
{
   cell_t c;
   begin_c_loop(c,t)
      C_PROFILE(c,t,nv) = USER INPUT2  ;
   end_c_loop(c,t)
}
```

Example 8 -Target Mass Flow Rate UDF as a Function of Physical Flow Time

For some unsteady problems, it is desirable that the target mass flow rate be a function of the physical flow time. This boundary condition can be applied using a DEFINE_PROFILE UDF. The following UDF, named tm_pout2, adjusts the mass flow rate from $1.00 kg/s$ to $1.35 kg/s$ when the physical time step is greater than 0.2 seconds. After it has been interpreted or compiled, you can activate this UDF in the Pressure Outlet boundary condition dialog box in ANSYS FLUENT by selecting the Specify target mass-flow rate option, and then choosing the UDF name from the corresponding drop-down list.

> **i** Note that the mass flow rate profile is a function of time and only one constant value should be applied to all zone faces at a given time.

```
/* UDF for setting target mass flow rate in pressure-outlet      */
/* at t<0.2 sec the target mass flow rate set to 1.00 kg/s       */
/* when t>0.2 sec the target mass flow rate will change to 1.35 kg/s */

#include "udf.h"
DEFINE_PROFILE(tm_pout2, t, nv)
{
  face_t f ;

  real flow_time = RP_Get_Real("flow-time");

  if (flow_time < 0.2 )
    {
      printf("Time           = %f  sec. \n",flow_time);
      printf("Targeted mass-flow rate set at 1.0 kg/s \n");

      begin_f_loop(f,t)
       {
         F_PROFILE(f,t,nv) = 1.0 ;
       }
      end_f_loop(f,t)
    }
  else
    {
      printf("Time           = %f  sec. \n",flow_time);
      printf("Targeted mass-flow rate set at 1.35 kg/s \n") ;


      begin_f_loop(f,t)
        {
```

```
            F_PROFILE(f,t,nv) = 1.35 ;
          }
      end_f_loop(f,t)
      }
}
```

Example 9 - Mass Flow Rate UDF for the Mass Flow Inlet

This UDF is used to provide a time-varying specification of the mass flow rate. This boundary condition can be applied using a DEFINE_PROFILE UDF. The following UDF, named inlet_mf, adjusts the mass flow rate from $1.00kg/s$ to $1.35kg/s$ when the physical time step is greater than 0.2 seconds. After it has been interpreted or compiled, you can activate this UDF in the Mass-Flow Inlet boundary condition dialog box in ANSYS FLUENT by selecting the UDF from the Mass Flow Rate drop-down list.

```
#include "udf.h"

DEFINE_PROFILE(inlet_mf,th,i)
{
 face_t f;
 begin_f_loop(f,th)
 {

 if(CURRENT_TIME <= 0.01)
  F_PROFILE(f,th,i) = 3.0;
 else if(CURRENT_TIME <=0.02 && CURRENT_TIME >0.01)
  F_PROFILE(f,th,i) = 4.0;
 else
  F_PROFILE(f,th,i) = 5.0;
 }
end_f_loop(f,th);
}
```

Hooking a Boundary Profile UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PROFILE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., vis_res) will become visible and selectable in the appropriate boundary condition or cell zone condition dialog box (e.g., the Velocity Inlet dialog box) in ANSYS FLUENT. See Section 6.2.15: Hooking DEFINE_PROFILE UDFs for details.

2.3.16 DEFINE_PROPERTY UDFs

Description

You can use DEFINE_PROPERTY to specify a custom material property in ANSYS FLUENT for single-phase and multiphase flows. When you are writing a user-defined mixing law UDF for a mixture material, you will need to use special utilities to access species material properties. These are described below. If you want to define a custom mass diffusivity property when modeling species transport, you must use DEFINE_DIFFUSIVITY instead of DEFINE_PROPERTY. See Section 2.3.3: DEFINE_DIFFUSIVITY for details on DEFINE_DIFFUSIVITY UDFs. For an overview of the ANSYS FLUENT solution process which shows when a DEFINE_PROPERTY UDF is called, refer to Figures 1.9.1, 1.9.2, and 1.9.3.

Some of the properties you can customize using DEFINE_PROPERTY are:

- density (as a function of temperature)

- viscosity

- thermal conductivity

- absorption and scattering coefficients

- laminar flow speed

- rate of strain

- user-defined mixing laws for density, viscosity, and thermal conductivity of mixture materials

- partially-premixed unburnt properties for unburnt density, unburnt temperature, unburnt specific heat, and unburnt thermal diffusivity

> **i** If you would like to use a UDF to define specific heat properties, you must use the DEFINE_SPECIFIC_HEAT, as described in Section 2.3.21: DEFINE_SPECIFIC_HEAT.

> **i** Note that when you specify a user-defined density function for a compressible liquid flow application, you must also include a speed of sound function in your model. Compressible liquid density UDFs can be used in the pressure-based solver and for single phase, multiphase mixture and cavitation models, only. See the example below for details.

For Multiphase Flows

- surface tension coefficient (VOF model)

- cavitation parameters including surface tension coefficient and vaporization pressure (Mixture, cavitation models)

- heat transfer coefficient (Mixture model)

- particle or droplet diameter (Mixture model)

- speed of sound function (Mixture, cavitation models)

- density (as a function of pressure) for compressible liquid flows only (Mixture, cavitation models)

- granular temperature and viscosity (Mixture, Eulerian models)

- granular bulk viscosity (Eulerian model)

- granular conductivity (Eulerian model)

- frictional pressure and viscosity (Eulerian model)

- frictional modulus (Eulerian model)

- elasticity modulus (Eulerian model)

- radial distribution (Eulerian model)

- solids pressure (Eulerian, Mixture models)

- diameter (Eulerian, Mixture models)

Usage

DEFINE_PROPERTY(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the property function is to be applied.

Function returns
real

There are three arguments to DEFINE_PROPERTY: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the real property *only* for a single cell and return it to the solver.

Note that like source term UDFs, property UDFs (defined using DEFINE_PROPERTY) are called by ANSYS FLUENT from within a loop on cell threads. The solver passes all of the variables needed to allow a DEFINE_PROPERTY UDF to define a custom material, since properties are assigned on a cell basis. Consequently, your UDF will *not* need to loop over cells in a zone since ANSYS FLUENT is already doing it.

Auxiliary Utilities

Some commonly-used auxiliary utilities for custom property UDFs are described below. They are generic_property, MATERIAL_PROPERTY, THREAD_MATERIAL, and mixture_species_loop.

generic_property is a general purpose function that returns the real value for the given property id for the given thread material. It is defined in prop.h and is used only for species properties.

The following Property_ID variables are available:

- PROP_rho, density

- PROP_mu, viscosity

- PROP_ktc, thermal conductivity

generic_property (name,c,t,prop,id,T)

Argument Type	Description
symbol name	Function name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which property function is to be applied.
Property *prop	Pointer to property array for the thread material that can be obtained through the macro MATERIAL_PROPERTY(m) See below.
Property_ID id	Property ID of the required property you want to define a custom mixing law for (e.g., PROP_ktc for thermal conductivity). See below for list of variables.
real T	Temperature at which the property is to be evaluated (used only if a polynomial method is specified).

Function returns
real

MATERIAL_PROPERTY is defined in `materials.h` and returns a real pointer to the `Property` array `prop` for the given material pointer `m`.

```
MATERIAL_PROPERTY(m)
```

Argument Type   **Description**
`Material *m`   Material pointer.

Function returns
`real`

THREAD_MATERIAL is defined in `threads.h` and returns real pointer `m` to the `Material` that is associated with the given cell thread `t`.

> **i**   Note that in previous versions of FLUENT, THREAD_MATERIAL took two arguments (`t,i`), but now only takes one (`t`).

```
THREAD_MATERIAL(t)
```

Argument Type   **Description**
`Thread *t`   Pointer to cell thread.

Function returns
`real`

mixture_species_loop is defined in `materials.h` and loops over all of the species for the given mixture material.

```
mixture_species_loop (m,sp,i)
```

Argument Type   **Description**
`Material *m`   Material pointer.
`Material *sp`   Species pointer.
`int i`   Species index.

Function returns
`real`

Example 1 - Temperature-dependent Viscosity Property

The following UDF, named `cell_viscosity`, generates a variable viscosity profile to simulate solidification. The function is called for every cell in the zone. The viscosity in the warm ($T > 288$ K) fluid has a molecular value for the liquid (5.5×10^{-3}kg/m-s), while the viscosity for the cooler region ($T < 286$ K) has a much larger value (1.0 kg/m-s). In the intermediate temperature range (286 K $\leq T \leq 288$ K), the viscosity follows a linear profile that extends between the two values given above:

$$\mu = 143.2135 - 0.49725T \qquad (2.3\text{-}8)$$

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The source code can be interpreted or compiled in ANSYS FLUENT.

```
/************************************************************************
   UDF that simulates solidification by specifying a temperature-
   dependent viscosity property
************************************************************************/
#include "udf.h"

DEFINE_PROPERTY(cell_viscosity,c,t)
{
  real mu_lam;
  real temp = C_T(c,t);

  if (temp > 288.)
    mu_lam = 5.5e-3;
  else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
  else
    mu_lam = 1.;

  return mu_lam;
}
```

The function `cell_viscosity` is defined on a cell. Two real variables are introduced: `temp`, the value of `C_T(c,t)`, and `mu_lam`, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of `mu_lam` is computed. At the end of the function the computed value for the viscosity (`mu_lam`) is returned to the solver.

Example 2 - User-defined Mixing Law for Thermal Conductivity

You can use DEFINE_PROPERTY to define custom user-defined mixing laws for density, viscosity, and conductivity of mixture materials. In order to access species material properties your UDF will need to utilize auxiliary utilities that are described above.

The following UDF, named mass_wtd_k, is an example of a mass-fraction weighted conductivity function. The UDF utilizes the generic_property function to obtain properties of individual species. It also makes use of MATERIAL_PROPERTY and THREAD_MATERIAL.

```
/***********************************************************************
  UDF that specifies a custom mass-fraction weighted conductivity
***********************************************************************/
#include "udf.h"

DEFINE_PROPERTY(mass_wtd_k,c,t)
{
    real sum = 0.; int i;
    Material *sp;
    real ktc;
    Property *prop;
    mixture_species_loop(THREAD_MATERIAL(t),sp,i)
      {
        prop = (MATERIAL_PROPERTY(sp));
        ktc = generic_property(c,t,prop,PROP_ktc,C_T(c,t));
        sum += C_YI(c,t,i)*ktc;
      }
    return sum;
}
```

Example 3 - Surface Tension Coefficient UDF

DEFINE_PROPERTY can also be used to define a surface tension coefficient UDF for the multiphase VOF model. The following UDF specifies a surface tension coefficient as a quadratic function of temperature. The source code can be interpreted or compiled in ANSYS FLUENT.

```
/****************************************************************
Surface Tension Coefficient UDF for the multiphase VOF Model
****************************************************************/

#include "udf.h"
DEFINE_PROPERTY(sfc,c,t)
```

```
{
    real T = C_T(c,t);
    return 1.35 - 0.004*T + 5.0e-6*T*T;
}
```

> *i* Note that surface tension UDFs for the VOF and Mixture multiphase mod-
> els are both hooked to ANSYS FLUENT in the Phase Interaction dialog box,
> but in different ways. For the VOF model, the function hook is located in
> the Surface Tension tab in the dialog box. For the Mixture model, however,
> the function hook is located in the Mass tab, and will become visible upon
> selecting the Cavitation option.

Example 4 - Density Function for Compressible Liquids

Liquid density is not a constant but is instead a function of the pressure field. In order to stabilize the pressure solution for compressible flows in ANSYS FLUENT, an extra term related to the speed of sound is needed in the pressure correction equation. Consequently, when you want to define a custom density function for a compressible flow, your model must also include a speed of sound function. Although you can direct ANSYS FLUENT to calculate a speed of sound function by choosing one of the available methods (e.g., piecewise-linear, polynomial) in the Create/Edit Materials dialog box, as a general guideline you should define a speed of sound function along with your density UDF using the formulation:

$$\sqrt{(\frac{\partial p}{\partial \rho})}$$

For simplicity, it is recommended that you concatenate the density and speed of sound functions into a single UDF source file.

The following UDF source code example contains two concatenated functions: a density function named `superfluid_density` that is defined in terms of pressure and a custom speed of sound function named `sound_speed`.

```
/************************************************************************
Density and speed of sound UDFs for compressible liquid flows.
For use with pressure-based solver, for single phase, multiphase mixture
or cavitation models only.
Note that for density function, dp is the difference between a cell
absolute pressure and reference pressure.
************************************************************************/
#include "udf.h"
```

```
#define BMODULUS 2.2e9
#define rho_ref 1000.0
#define p_ref 101325

DEFINE_PROPERTY(superfluid_density, c, t)
{
    real rho;
    real p, dp;
    real p_operating;

    p_operating = RP_Get_Real ("operating-pressure");

    p = C_P(c,t) + p_operating;
    dp = p-p_ref;
    rho = rho_ref/(1.0-dp/BMODULUS);
    return rho;
}


DEFINE_PROPERTY(sound_speed, c,t)

{
    real a;
    real p, dp,p_operating;

    p_operating = RP_Get_Real ("operating-pressure");

    p = C_P(c,t) + p_operating;
    dp = p-p_ref;
    a = (1.-dp/BMODULUS)*sqrt(BMODULUS/rho_ref);
    return a;
}
```

Hooking a Property UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_PROPERTY is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., sound_speed) will become visible and selectable in dialog boxes in ANSYS FLUENT.
See Section 6.2.16: Hooking DEFINE_PROPERTY UDFs for details.

2.3.17 DEFINE_SCAT_PHASE_FUNC

Description

You can use DEFINE_SCAT_PHASE_FUNC to specify the radiation scattering phase function for the discrete ordinates (DO) model. The function computes two values: the fraction of radiation energy scattered from direction i to direction j, and the forward scattering factor.

Usage

DEFINE_SCAT_PHASE_FUNC(name,cosine,f)

Argument Type	Description
symbol name	UDF name.
real cosine	Cosine of the angle between directions i and j.
real *f	Pointer to the location in memory where the real forward scattering factor is stored.

Function returns
real

There are three arguments to DEFINE_SCAT_PHASE_FUNC: name, cosine, and f. You supply name, the name of the UDF. cosine and f are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the real fraction of radiation energy scattered from direction i to direction j and return it to the solver. Note that the solver computes and stores a scattering matrix for each material by calling this function for each unique pair of discrete ordinates.

Example

In the following example, a number of UDFs are concatenated in a single C source file. These UDFs implement backward and forward scattering phase functions that are cited by Jendoubi et al. [5]. The source code can be interpreted or compiled in ANSYS FLUENT.

```
/*********************************************************************
    UDFs that implement backward and forward scattering
    phase functions as cited by Jendoubi et. al.
*********************************************************************/

#include "udf.h"

DEFINE_SCAT_PHASE_FUNC(ScatPhiB2,c,fsf)
{
  real phi=0;
  *fsf = 0;
  phi = 1.0 - 1.2*c + 0.25*(3*c*c-1);
  return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiB1,c,fsf)
{
  real phi=0;
  *fsf = 0;
  phi = 1.0 - 0.56524*c + 0.29783*0.5*(3*c*c-1) +
    0.08571*0.5*(5*c*c*c-3*c) + 0.01003/8*(35*c*c*c*c-30*c*c+3) +
    0.00063/8*(63*c*c*c*c*c-70*c*c*c+15*c);
  return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF3,c,fsf)
{
  real phi=0;
  *fsf = 0;
  phi = 1.0 + 1.2*c + 0.25*(3*c*c-1);
  return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF2,c,fsf)
{
  real phi=0;
  real coeffs[9]={1,2.00917,1.56339,0.67407,0.22215,0.04725,
                  0.00671,0.00068,0.00005};
  real P[9];
  int i;
  *fsf = 0;
  P[0] = 1;
  P[1] = c;
```

```
   phi = P[0]*coeffs[0] + P[1]*coeffs[1];
   for(i=1;i<7;i++)
     {
       P[i+1] = 1/(i+1.0)*((2*i+1)*c*P[i] - i*P[i-1]);
       phi += coeffs[i+1]*P[i+1];
     }
   return (phi);
}


DEFINE_SCAT_PHASE_FUNC(ScatIso,c,fsf)
{
  *fsf=0;
  return (1.0);
}
```

Hooking a Scattering Phase UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_SCAT_PHASE_FUNCTION is interpreted
(Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name
that you specified in the DEFINE macro argument (e.g., ScatPhiB) will become visible
and selectable in the **Create/Edit Materials** dialog box in ANSYS FLUENT.
See Section 6.2.17: Hooking DEFINE_SCAT_PHASE_FUNC UDFs for details.

2.3.18 DEFINE_SOLAR_INTENSITY

Description

You can use the DEFINE_SOLAR_INTENSITY macro to define direct solar intensity or diffuse solar intensity UDFs for the solar load model. See Chapter 13: Modeling Heat Transfer in the separate User's Guide for more information on the solar load model.

i Note that solar intensity UDFs are used with the Solar Model, which is available only for the 3d geometries in ANSYS FLUENT.

Usage

DEFINE_SOLAR_INTENSITY(name,sum_x,sun_y,sun_z,S_hour,S_minute)

Argument Type	Description
symbol name	UDF name.
real sun_x	x component of the sun direction vector.
real sun_y	y component of the sun direction vector.
real sun_z	z component of the sun direction vector.
real S_hour	Time in hours.
real S_minute	Time in minutes.

Function returns
real

There are six arguments to DEFINE_SOLAR_INTENSITY: name, sun_x, sun_y, sun_z, S_hour, and S_minute. You provide the name of your user-defined function. The variables sun_x, sun_y, sun_z, S_hour, and S_minute are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the direct or diffuse solar irradiation and return the real value (in w/m^2) to the solver.

Example

The following source code contains two UDFs: sol_direct_intensity computes the direct solar irradiation and returns it to the ANSYS FLUENT solver, and sol_diffuse_intensity computes the diffuse solar irradiation.

```
#include "udf.h"

DEFINE_SOLAR_INTENSITY(sol_direct_intensity,sun_x,sun_y,sun_z,hour,minute)
{
  real intensity;
  intensity = 1019;
  printf("solar-time=%f intensity=%e\n", minute, intensity);
  return intensity;
}

DEFINE_SOLAR_INTENSITY(sol_diffuse_intensity,sun_x,sun_y,sun_z,hour,minute)
{
  real intensity;
  intensity = 275;
  printf("solar-time=%f intensity-diff=%e\n", minute, intensity);
  return intensity;
}
```

Hooking a Solar Intensity UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_SOLAR_INTENSITY` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name that you specified (e.g., `sol_direct_intensity`) in the `DEFINE` macro argument will become visible and selectable for Direct Solar Irradiation and Diffuse Solar Irradiation in the Radiation Model dialog box in ANSYS FLUENT. Note that the solar load model must be enabled. See Section 6.2.18: Hooking `DEFINE_SOLAR_INTENSITY` UDFs for details.

2.3.19 DEFINE_SOURCE

Description

You can use DEFINE_SOURCE to specify custom source terms for the different types of solved transport equations in ANSYS FLUENT (except the discrete ordinates radiation model) including:

- mass

- momentum

- k, ϵ

- energy (also for solid zones)

- species mass fractions

- P1 radiation model

- user-defined scalar (UDS) transport

- granular temperature (Eulerian, Mixture multiphase models)

Usage

DEFINE_SOURCE(name,c,t,dS,eqn)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies cell on which the source term is to be applied.
Thread *t	Pointer to cell thread.
real dS[]	Array that contains the derivative of the source term with respect to the dependent variable of the transport equation.
int eqn	Equation number.

Function returns
real

There are five arguments to `DEFINE_SOURCE`: `name`, `c`, `t`, `dS`, and `eqn`. You supply `name`, the name of the UDF. `c`, `t`, `dS`, and `eqn` are variables that are passed by the ANSYS FLUENT solver to your UDF. Note that the source term derivatives may be used to linearize the source term if they enhance the stability of the solver. To illustrate this, note that the source term can be expressed, in general, as Equation 2.3-9, where ϕ is the dependent variable, A is the explicit part of the source term, and $B\phi$ is the implicit part.

$$S_\phi = A + B\phi \qquad (2.3\text{-}9)$$

Specifying a value for B in Equation 2.3-9 can enhance the stability of the solution and help convergence rates due to the increase in diagonal terms on the solution matrix. ANSYS FLUENT automatically determines if the value of B that is given by the user will aid stability. If it does, then ANSYS FLUENT will define A as $S^* - (\partial S/\partial \phi)^* \phi^*$, and B as $(\partial S/\partial \phi)^*$. If not, the source term is handled explicitly.

Your UDF will need to compute the `real` source term *only* for a single cell and return the value to the solver, but you have the choice of setting the implicit term `dS[eqn]` to $dS/d\phi$, or forcing the explicit solution of the source term by setting it equal to 0.0.

Note that like property UDFs, source term UDFs (defined using `DEFINE_SOURCE`) are called by ANSYS FLUENT from within a loop on cell threads. The solver passes to the `DEFINE_SOURCE` term UDF all the necessary variables it needs to define a custom source term, since source terms are solved on a cell basis. Consequently, your UDF will *not* need to loop over cells in the thread since ANSYS FLUENT is already doing it.

The units on all source terms are of the form generation-rate/volume. For example, a source term for the continuity equation would have units of kg/m^3-s.

Example 1 - Source Term Addition

The following UDF, named `xmom_source`, is used to add source terms in ANSYS FLUENT. The source code can be interpreted or compiled. The function generates an x-momentum source term that varies with y position as

$$\text{source} = -0.5 C_2 \rho y |v_x| v_x$$

Suppose

$$\text{source} = S = -A|v_x|v_x$$

where

$$A = 0.5 C_2 \rho y$$

Then

$$\frac{dS}{dv_x} = -A|v_x| - Av_x \frac{d}{dv_x}(|v_x|)$$

The source term returned is

$$\text{source} = -A|v_x|v_x$$

and the derivative of the source term with respect to v_x (true for both positive and negative values of v_x) is

$$\frac{dS}{dv_x} = -2A|v_x|$$

```
/********************************************************************/
/* UDF for specifying an x-momentum source term in a spatially     */
/* dependent porous media                                          */
/********************************************************************/

#include "udf.h"

#define C2 100.0

DEFINE_SOURCE(xmom_source,c,t,dS,eqn)
{
  real x[ND_ND];
  real con, source;

  C_CENTROID(x,c,t);
  con = C2*0.5*C_R(c,t)*x[1];

  source = -con*fabs(C_U(c, t))*C_U(c,t);
  dS[eqn] = -2.*con*fabs(C_U(c,t));

  return source;
}
```

Example 2 - Degassing Boundary Condition

The following UDFs are used to define the bottom surface as a standard velocity inlet for the gas (primary) phase. The inlet VOF of the droplet phase is 0 and a negative source term for secondary phase mass conservation is set for the layer of cells next to the outlet. The source removes all secondary phase mass in the cell during one time step. The recoil force due to the mass source is also calculated.

```
/*******************************************************************/
/*This UDF is an implementation of the degassing boundary condition*/
/*******************************************************************/

#include "udf.h"
#include "sg.h"
#include "sg_mphase.h"
#include "flow.h"
#include "mem.h"
#include "metric.h"

DEFINE_SOURCE(degassing_source, cell, thread, dS, eqn)
{
  real source;
  Thread *tm = THREAD_SUPER_THREAD(thread);

  source =  -C_R(cell,thread)*C_VOF(cell,thread)/CURRENT_TIMESTEP ;

  C_UDMI(cell,tm,0) = source;

  dS[eqn] =  -C_R(cell,thread)/CURRENT_TIMESTEP;

  return source;
}

DEFINE_SOURCE(x_prim_recoil, cell, tp, dS, eqn)
{
  real source;
  Thread *tm = THREAD_SUPER_THREAD(tp);
  Thread *ts;

  ts = THREAD_SUB_THREAD(tm,1);

  source =  -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_U(cell,tp);

  dS[eqn] =-C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
```

```
    return source;
  }


  DEFINE_SOURCE(x_sec_recoil, cell, ts, dS, eqn)
  {
    real source;
    Thread *tm = THREAD_SUPER_THREAD(ts);

    source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_U(cell,ts);

    dS[eqn] =  -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;

    return source;
  }

  DEFINE_SOURCE(y_prim_recoil, cell, tp, dS, eqn)
  {
    real source;
    Thread *tm = THREAD_SUPER_THREAD(tp);
    Thread *ts;

    ts = THREAD_SUB_THREAD(tm,1);

    source =  -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_V(cell,tp);

    dS[eqn] =-C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;

    return source;
  }


  DEFINE_SOURCE(y_sec_recoil, cell, ts, dS, eqn)
  {
    real source;
    Thread *tm = THREAD_SUPER_THREAD(ts);

    source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_V(cell,ts);

     dS[eqn] =  -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;

    return source;
```

```
}

DEFINE_SOURCE(z_prim_recoil, cell, tp, dS, eqn)
{
  real source;
  Thread *tm = THREAD_SUPER_THREAD(tp);
  Thread *ts;

  ts = THREAD_SUB_THREAD(tm,1);

  source =  -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_W(cell,tp);

  dS[eqn] =-C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;

  return source;
}


DEFINE_SOURCE(z_sec_recoil, cell, ts, dS, eqn)
{
  real source;
  Thread *tm = THREAD_SUPER_THREAD(ts);

  source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_W(cell,ts);

   dS[eqn] =  -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;

  return source;
}
```

Hooking a Source UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_SOURCE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., xmom_source) will become visible and selectable in the Fluid or Solid cell zone condition dialog box in ANSYS FLUENT. See Section 6.2.19: Hooking DEFINE_SOURCE UDFs for details.

2.3.20 DEFINE_SOX_RATE

Description

You can use DEFINE_SOX_RATE to specify a custom SO_x rate that can either replace the internally-calculated SO_x rate in the source term equation, or be added to the ANSYS FLUENT rate. Example 1 demonstrates this use of DEFINE_SOX_RATE. The default functionality is to add user-defined rates to the ANSYS FLUENT-calculated rates. If the Replace with UDF Rate option is enabled in the SOx Model dialog box, then the ANSYS FLUENT-calculated rate will not be used and it will instead be replaced by the SO_x rate you have defined in your UDF. When you hook a SO_x rate UDF to the graphical interface without checking the Replace with UDF Rate box, then the user-defined SO_x rate will be *added* to the internally-calculated rate for the source term calculation.

DEFINE_SOX_RATE may also be used to calculate the upper limit for the integration of the temperature PDF (when temperature is accounted for in the turbulence interaction modeling). You can calculate a custom maximum limit (T_{max}) for each cell and then assign it to the POLLUT_CTMAX(Pollut_Par) macro (see Section 3.2.7: SO_x Macros for further details about data access macros). Example 2 demonstrates this use of DEFINE_SOX_RATE.

i If you want to use DEFINE_SOX_RATE only for the purpose of specifying T_{max}, be sure that the user-defined SO_x rate does not alter the internally-calculated rate for the source term calculation.

Usage

DEFINE_SOX_RATE(name,c,t,Pollut,Pollut_Par, SOx)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the SO_x rate is to be applied.
Pollut_Cell *Pollut	Pointer to the data structure that contains the common data at each cell.
Pollut_Parameter *Pollut_Par	Pointer to the data structure that contains auxiliary data.
SOx_Parameter *SOx	Pointer to the data structure that contains data specific to the SOx model.

Function returns
void

There are six arguments to `DEFINE_SOX_RATE`: `name`, `c`, `t`, `Pollut`, `Pollut_Par` and `SOx`. You will supply `name`, the name of the UDF. `c`, `t`, `Pollut`, `Pollut_Par` and `SOx` are variables that are passed by the ANSYS FLUENT solver to your function. A `DEFINE_SOX_RATE` function does not output a value. The calculated SO_2 rates (or other pollutant species rates) are returned through the `Pollut` structure as the forward rate `POLLUT_FRATE(Pollut)` and reverse rate `POLLUT_RRATE(Pollut)`, respectively.

> **i** The data contained within the SO_x structure is specific *only* to the SO_x model. Alternatively, the `Pollut` structure contains data at each cell that is useful for all pollutant species (e.g., forward and reverse rates, gas phase temperature, density). The `Pollut_Par` structure contains auxiliary data common for all pollutant species (e.g. equation solved, universal gas constant, species molecular weights). Note that molecular weights extracted from the `Pollut_Par` structure (i.e., `Pollut_Par->sp[IDX(i)].mw` for pollutant species—NO, HCN, etc.—and `Pollut_Par->sp[i].mw` for other species, such as O_2) has units of kg/kg-mol.

Example 1

The following compiled UDF, named `user_sox`, computes the rates for SO_2 and SO_3 formation according to the reaction given in Equation 2.3-10. Note that this UDF will replace the ANSYS FLUENT rate *only* if you select the **Replace with UDF Rate** option in the **SOx Model** dialog box. Otherwise, the rate computed in the UDF will be added to ANSYS FLUENT's default rate. See Section 6.2.20: Hooking `DEFINE_SOX_RATE` UDFs for details.

It is assumed that the release of fuel sulfur from fuel is proportional to the rate of release of volatiles and all sulfur is in the form of SO_2 when released to the gas phase. The reversible reaction for SO_2/SO_3 is given below:

$$SO_3 + O \longleftrightarrow SO_2 + O_2 \tag{2.3-10}$$

with forward and reverse rates of reaction (k_f and k_r, respectively) in the Arrhenius form

$k_f = 1.2e^6 e^{(-39765.575/RT)}$

$k_r = 1.0e^4 T^{-1} e^{(-10464.625/RT)}$

The O atom concentration in the gas phase (o_{eq}) is computed using the partial equilibrium assumption, which states

$o_{eq} = 36.64 T^{0.5} e^{(-27123.0/RT)} \sqrt{[O_2]}$

where $[O_2]$ is the molar concentration of oxygen. Here, all units are in m-gmol-J-sec.

The function `so2_so3_rate` is used to compute the forward and reverse rates for both SO_2 and SO_3.

The rate of release of SO_2 from volatiles $(S_{SO_2,\text{volatile}})$ is given by:

$$S_{SO_2,\text{volatile}} = \frac{1000 r_{\text{volatile}} Y_{S,\text{volatile}} Y_{S,SO_2}}{M_{w,S} V}$$

where r_{volatile} is the rate of release of volatiles in kg/sec, $Y_{S,\text{volatile}}$ is the mass fraction of sulfur species in volatiles, Y_{S,SO_2} is the mass fraction of fuel S that converts to SO_2, $M_{w,S}$ is the molecular weight of sulfur in kg/kg-mol, and V is the cell volume in m^3.

See Section 3.2.7: SO_x Macros for details about the SO_x macros (e.g., `POLLUT_EQN`, `MOLECON`, `ARRH`) that are used in pollutant rate calculations in this UDF.

```
/********************************************************************
   UDF example of User-Defined SOx Rate for ANSYS FLUENT 12 or later

   If used with the "Replace with UDF" radio button activated,
   this UDF will replace the default fluent SOx rates.

   The flag "Pollut_Par->pollut_io_pdf == IN_PDF" should always
   be used for rates other than that from char N, so that if
   requested, the contributions will be PDF integrated. Any
   contribution from char must be included within a switch
   statement of the form "Pollut_Par->pollut_io_pdf == OUT_PDF".
   *
   * Arguments:
   *    char sox_func_name           - UDF name
   *    cell_t c                     - Cell index
   *    Thread *t                    - Pointer to cell thread on
   *                                   which the SOx rate is to be
   *                                   applied
   *    Pollut_Cell *Pollut          - Pointer to Pollut structure
   *    Pollut_Parameter *Pollut_Par - Pointer to Pollut_Par
   *                                   structure
   *    SOx_Parameter *SOx           - Pointer to SOx structure

   ********************************************************************/

#include "udf.h"

static void so2_so3_rate(cell_t c, Thread* t, Pollut_Cell *Pollut,
                Pollut_Parameter *Pollut_Par, SOx_Parameter *SOx);
```

```
DEFINE_SOX_RATE(user_sox, c, t, Pollut, Pollut_Par, SOx)
{
  POLLUT_FRATE(Pollut) = 0.0;
  POLLUT_RRATE(Pollut) = 0.0;

  switch (Pollut_Par->pollut_io_pdf) {
  case IN_PDF:
    /* Included source terms other than those from char */

    if (SOx->user_replace) {
      /* This rate replaces the default ANSYS FLUENT rate */
      so2_so3_rate(c,t,Pollut,Pollut_Par,SOx);
    }
    else {
      /* This rate is added to the default ANSYS FLUENT rate */
      so2_so3_rate(c,t,Pollut,Pollut_Par,SOx);
    }
    break;
  case OUT_PDF:
    /* Char Contributions, must be included here */
    break;

  default:
    /* Not used */
    break;
  }

}

static void so2_so3_rate(cell_t c, Thread* t, Pollut_Cell *Pollut,
                Pollut_Parameter *Pollut_Par, SOx_Parameter *SOx)
{
  /* Pollut_Par->nfstreams = Number of fuel streams
   * Pollut->r_fuel_gls[i] = Rate of volatile release for stream "i"
   *                         per unit volume in kg/m3-sec
   * SOx->Ys_fuelvolat[i] = Mass fraction of S in volatile stream "i"
   * SOx->fuels_so2_frac[i] = Partition fraction of SO2 in stream "i"
   */
  real kf,kr,rf=0,rr=0;
  real o_eq;
  real r_volatile,Ys_volatile,fuels_so2_frac;
```

```
   Rate_Const K_F = {1.2e6,  0.0, 39765.575};
   Rate_Const K_R = {1.0e4, -1.0, 10464.625};
   Rate_Const K_O = {36.64,  0.5, 27123.0};

   /* SO3 + O <-> SO2 + O2 */
   kf = ARRH(Pollut, K_F);
   kr = ARRH(Pollut, K_R);

   o_eq  = ARRH(Pollut, K_O)*sqrt(MOLECON(Pollut, O2));

   if (POLLUT_EQN(Pollut_Par) == EQ_SO2) {
     int ifstream;

     Ys_volatile = 1.e-04;
     fuels_so2_frac = 1.;

     for(ifstream=0; ifstream<Pollut_Par->nfstreams; ifstream++) {
       rf += Pollut->r_fuel_gls[ifstream]*SOx->Ys_fuelvolat[ifstream]
           *SOx->fuels_so2_frac[ifstream]*1000./Pollut_Par->sp[S].mw;
     }
     rf += kf*o_eq*MOLECON(Pollut, IDX(SO3));
     rr = -kr*MOLECON(Pollut, O2)*MOLECON(Pollut, IDX(SO2));
   }
   else if (POLLUT_EQN(Pollut_Par) == EQ_SO3) {
     rf = kr*MOLECON(Pollut, O2)*MOLECON(Pollut, IDX(SO2));
     rr = -kf*o_eq*MOLECON(Pollut, IDX(SO3));
   }

   POLLUT_FRATE(Pollut) += rf;
   POLLUT_RRATE(Pollut) += rr;
}
```

Example 2

The following compiled UDF, named sox_func_name, specifies a custom maximum limit (T_{max}) for the integration of the temperature PDF for each cell. Note that this UDF does not alter the internally-calculated SO_x rate.

See Section 3.2.7: SO_x Macros for details about the SO_x macro (POLLUT_CTMAX) used in this UDF.

```
/************************************************************
   UDF example of User-Defined Tmax value
```

```
 *
 * Arguments:
 *   char sox_func_name          - UDF name
 *   cell_t c                    - Cell index
 *   Thread *t                   - Pointer to cell thread
 *                                 on which the SOx rate
 *                                 is to be applied
 *   Pollut_Cell *Pollut         - Pointer to Pollut_Cell
 *                                 structure
 *   Pollut_Parameter *Pollut_Par - Pointer to Pollut_Parameter
 *                                 structure
 *   SOx_Parameter *SOx          - Pointer to SOx_Parameter
 *                                 structure

   ANSYS FLUENT Version: 12.0 or later
*************************************************************/

#include "udf.h"

int ud_sox_do_once=1;

enum
{
  CELL_TMAX=0,
  N_REQUIRED_UDM
};

/*Compute/assign Tmax at each cell*/
real ud_eval_cell_tmax(cell_t c,Thread *t)
{
  real tmax = 0.;

  /* Compute cell-based Tmax value */
  tmax = 1.1*C_T(c,t); /* This is only an example */

  return tmax;
}

DEFINE_SOX_RATE(user_sox, c, t, Pollut, Pollut_Par, SOx)
{
  /* Assign cell-based Tmax value */
  POLLUT_CTMAX(Pollut_Par) = ud_eval_cell_tmax(c,t);
```

```
  /*POLLUT_CTMAX(Pollut_Par) = C_UDMI(c,t,CELL_TMAX);*/
}

DEFINE_ON_DEMAND(init_tmax)
{
  Domain *domain;
  register Thread *t;
  register cell_t c;

  Message("Computing/Storing cell Tmax values\n");
  domain = Get_Domain(1);

  /* Store User-Defined Tmax at each cell */
  if(ud_sox_do_once == 1) {
    if(n_udm < N_REQUIRED_UDM)
      Error("Not enough udm allocated\n");

    thread_loop_c (t,domain)
      begin_c_loop (c,t)
        C_UDMI(c,t,CELL_TMAX) = ud_eval_cell_tmax(c,t);
      end_c_loop (c,t)
    ud_sox_do_once = 0;
  }
  Message("Computing cell Tmax values completed..\n");
}
```

Hooking a SO$_x$ Rate UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_SOX_RATE is compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_sox) will become visible and selectable in the SOx Model dialog box in ANSYS FLUENT. See Section 6.2.20: Hooking DEFINE_SOX_RATE UDFs for details.

2.3.21 `DEFINE_SPECIFIC_HEAT`

Description

The `DEFINE_SPECIFIC_HEAT` macro can be used to define temperature dependent functions for specific heat and sensible enthalpy for fluid, solid and mixture materials (this does not include DPM particles). These functions must be defined in a consistent manner, that is, the enthalpy function should be the temperature integral of the specific heat function.

i This option is not available with the premixed, non-premixed and partially premixed models, and should be used as a compiled UDF only.

Usage

`DEFINE_SPECIFIC_HEAT(name, T, Tref, h, yi)`

Argument Type	Description
`symbol name`	UDF name
`real T`	Temperature for the calculation of the specific heat and enthalpy
`real Tref`	Reference temperature for the enthalpy calculation
`real *h`	Pointer to `real`
`real *yi`	Pointer to array of mass fractions of gas phase species

Function returns
`real`

There are five arguments to `DEFINE_SPECIFIC_HEAT`: `name`, `T`, `Tref`, `h`, and `yi`. You supply `name`, the name of the UDF. `T` and `Tref` are `real` variables that are passed by the ANSYS FLUENT solver to the UDF, and `h` is a pointer to `real`.

The UDF needs to return the `real` value of the specific heat, and set the sensible enthalpy to the value referenced by the `real` pointer `h`. Note that the entropy is not computed in the UDF, instead ANSYS FLUENT sets the entropy as $S = cp(Tmean)log(T/Tref)$, where cp is computed by the UDF at $Tmean$, and $Tmean$ is the mean logarithmic average of `T` and `Tref`.

Example

```
/************************************************************************
   UDF that computes specific heat and sets the sensible enthalpy
   to the referenced value
*************************************************************************/

#include "udf.h"
DEFINE_SPECIFIC_HEAT(my_user_cp, T, Tref, h, yi)
{
real cp=2000.;
*h = cp*(T-Tref);
return cp;
}
```

Hooking a Specific Heat UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_SPECIFIC_HEAT` is compiled (Chapter 5: Compiling UDFs), the name that you specified in the `DEFINE` macro argument (e.g., `my_user_cp`) will become visible and selectable in the Create/Edit Materials dialog box in ANSYS FLUENT. See Section 6.2.21: Hooking `DEFINE_SPECIFIC_HEAT` UDFs for details.

2.3.22 `DEFINE_SR_RATE`

Description

You can use `DEFINE_SR_RATE` to specify a custom surface reaction rate. A custom surface reaction rate function defined using this macro will overwrite the default reaction rate (e.g., `finite-rate`) that is specified in the Create/Edit Materials dialog box. A `DEFINE_VR_RATE` UDF is compatible with the laminar finite-rate model, but you must make sure that the stiff chemistry option is disabled.

An example of a reaction rate that depends upon gas species mass fractions is provided below. Also provided is a reaction rate UDF that takes into account site species.

i   Note that the three types of surface reaction species are internally numbered with an (integer) index `i` in order

Usage

```
DEFINE_SR_RATE(name,f,t,r,my,yi,rr)
```

Argument Type	Description
symbol name	UDF name.
face_t f	Index that identifies a face within the given thread (or cell in the case of surface reaction in a porous zone).
Thread *t	Pointer to face thread on which the surface rate reaction is to be applied.
Reaction *r	Pointer to data structure for the reaction.
real *mw	Pointer to array of species molecular weights.
real *yi	Pointer to array of mass fractions of gas species at the surface and the coverage of site species (or site fractions).
real *rr	Pointer to reaction rate.

Function returns
```
void
```

There are seven arguments to DEFINE_SR_RATE: name, f, t, r, my, yi, and rr. You supply name, the name of the UDF. After your UDF is compiled and linked, the name that you have chosen for your function will become visible and selectable in the graphical user interface in ANSYS FLUENT. f, t, r, my, and yi are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the reaction rate to the value referenced by the real pointer rr as shown in the examples below.

Example 1 - Surface Reaction Rate Using Species Mass Fractions

The following compiled UDF, named arrhenius, defines a custom surface reaction rate using species mass fractions in ANSYS FLUENT.

```
/*********************************************************************
   Custom surface reaction rate UDF
 *********************************************************************/
#include "udf.h"
/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0

real arrhenius_rate(real temp)
{
  return
```

```
    PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*temp)));
}

/* Species numbers. Must match order in ANSYS FLUENT dialog box */
#define HF  0
#define WF6 1
#define H2O 2
#define NUM_SPECS 3

/* Reaction Exponents */
#define HF_EXP  2.0
#define WF6_EXP 0.0
#define H2O_EXP 0.0

#define MW_H2 2.0
#define STOIC_H2 3.0

/* Reaction Rate Routine */

real reaction_rate(cell_t c, Thread *cthread,real mw[],real yi[])

/* Note that all arguments in the reaction_rate function
call in your .c source file MUST be on the same line or a
compilation error will occur */

{
  real concenHF = C_R(c,cthread)*yi[HF]/mw[HF];
  return arrhenius_rate(C_T(c,cthread))*pow(concenHF,HF_EXP);
}

DEFINE_SR_RATE(arrhenius,f,fthread,r,mw,yi,rr)
{
 *rr =
reaction_rate(F_C0(f,fthread),THREAD_T0(fthread),mw,yi);
}
```

Example 2 - Surface Reaction Rate Using Site Species

The following compiled UDF, named my_rate, defines a custom surface reaction rate that takes into account site species.

```
/********************************************************************
    Custom surface reaction rate UDF
 ********************************************************************/
#include "udf.h"
DEFINE_SR_RATE(my_rate,f,t,r,mw,yi,rr)
{
 Thread *t0=t->t0;
 cell_t c0=F_C0(f,t);
 real sih4  = yi[0];   /* mass fraction of sih4 at the wall */
 real si2h6 = yi[1];
 real sih2  = yi[2];
 real h2    = yi[3];
 real ar    = yi[4];   /* mass fraction of ar at the wall */

 real rho_w = 1.0, site_rho = 1.0e-6, T_w = 300.0;

 real si_s  = yi[6];   /* site fraction of si_s*/
 real sih_s = yi[7];   /* site fraction of sih_s*/

 T_w = F_T(f,t);
 rho_w = C_R(c0,t0)*C_T(c0,t0)/T_w;

 sih4  *= rho_w/mw[0];  /* converting of mass fractions
to molar concentrations */
 si2h6 *= rho_w/mw[1];
 sih2  *= rho_w/mw[2];
 h2    *= rho_w/mw[3];
 ar    *= rho_w/mw[4];

 si_s   *= site_rho;   /* converting of site fractions to
 site concentrations */
 sih_s  *= site_rho;

 if (STREQ(r->name, "reaction-1"))
   *rr = 100.0*sih4;
 else if (STREQ(r->name, "reaction-2"))
   *rr = 0.1*sih_s;
 else if (STREQ(r->name, "reaction-3"))
   *rr = 100*si2h6*si_s;
```

```
  else if (STREQ(r->name, "reaction-4"))
    *rr = 1.0e10*sih2;


}
```

Hooking a Surface Reaction Rate UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_SR_RATE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., my_rate) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.22: Hooking DEFINE_SR_RATE UDFs for details.

2.3.23 DEFINE_TRANS **UDFs**

The following DEFINE macros can be used to specify transition correlations for the Transition SST model in ANSYS FLUENT.

DEFINE_TRANS_FLENGTH

Description

You can use DEFINE_TRANS_FLENGTH to specify the transition length for the Transition SST turbulence model.

Usage

DEFINE_TRANS_FLENGTH(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index of cell on which the transition length function is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_TRANS_FLENGTH: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value for the transition length function to the solver.

Example

An example of a TRANS_FLENGTH UDF is provided at the end of this section.

Hooking a Transition Correlation UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_TRANS_FLENGTH is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the DEFINE macro argument (e.g., user_Flength) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.23: Hooking DEFINE_TRANS UDFs for details.

DEFINE_TRANS_RETHETA_C

Description

You can use DEFINE_TRANS_RETHETA_C to specify the critical momentum thickness Reynolds number for the Transition SST turbulence model.

Usage

DEFINE_TRANS_RETHETA_C(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the critical momentum thickness Reynolds number is to be applied.
Thread *t	Pointer to cell thread.

Function returns
real

There are three arguments to DEFINE_TRANS_RETHETA_C: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value for the critical momentum thickness Reynolds number to the solver.

Example

An example of a TRANS_RETHETA_C UDF is provided at the end of this section.

Hooking a Transition Correlation **UDF to** ANSYS FLUENT

After the UDF that you have defined using `DEFINE_TRANS_RETHETA_C` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the `DEFINE` macro argument (e.g., `user_Re_thetac`) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.23: Hooking `DEFINE_TRANS` UDFs for details.

`DEFINE_TRANS_RETHETA_T`

Description

You can use `DEFINE_TRANS_RETHETA_T` to specify the transition onset momentum thickness Reynolds number for the Transition SST turbulence model.

Usage

`DEFINE_TRANS_RETHETA_T(name,c,t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the transition onset momentum thickness Reynolds number is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_TRANS_RETHETA_T`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the `real` value for the transition onset momentum thickness Reynolds number to the solver.

Example

The following functions (which are concatenated into a single C source code file) demonstrate this usage.

```
#include "udf.h"

DEFINE_TRANS_FLENGTH(user_Flength, c, t)
{
  real Flength = 31.468;

  return Flength;
}

DEFINE_TRANS_RETHETA_C(user_Re_thetac, c, t)
{
  real Re_thetac = 176.396;

  return Re_thetac;
}

DEFINE_TRANS_RETHETA_T(user_Re_thetat, c, t)
{
  real Re_thetat = 210;

  return Re_thetat;
}
```

Hooking a Transition Correlation UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_TRANS is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the DEFINE macro argument (e.g., user_Re_thetat) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.23: Hooking DEFINE_TRANS UDFs for details.

2.3.24 DEFINE_TURB_PREMIX_SOURCE

Description

You can use DEFINE_TURB_PREMIX_SOURCE to customize the turbulent flame speed and source term in the premixed combustion model (see Chapter 17: Modeling Premixed Combustion in the separate User's Guide) and the partially premixed combustion model (see Chapter 18: Modeling Partially Premixed Combustion in the separate User's Guide).

Usage

DEFINE_TURB_PREMIX_SOURCE(name,c,t,turb_flame_speed,source)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the turbulent premixed source term is to be applied.
real *turb_flame_speed	Pointer to the turbulent flame speed.
real *source	Pointer to the reaction progress source term.

Function returns
void

There are five arguments to DEFINE_TURB_PREMIX_SOURCE: name, c, t, turb_flame_speed, and source. You supply name, the name of the UDF. c, t, turb_flame_speed, and source are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the turbulent flame speed to the value referenced by the turb_flame_speed pointer. It will also need to set the source term to the value referenced by the source pointer.

Example

The following UDF, named turb_flame_src, specifies a custom turbulent flame speed and source term in the premixed combustion model. The source code must be executed as a compiled UDF in ANSYS FLUENT.

In the standard premixed combustion model in ANSYS FLUENT, the mean reaction rate of the progress variable (that is, the source term) is modeled as

$$\rho S_c = \rho_u U_t |\nabla c| \tag{2.3-11}$$

where c is the mean reaction progress variable, ρ is the density, and U_t is the turbulent flame speed.

In the UDF example, the turbulent flame speed is modeled as

$$U_t = U_l\sqrt{1 + (u'/U_l)^2} \tag{2.3-12}$$

where U_l is the laminar flame speed and u' is the turbulent fluctuation. Note that the partially premixed combustion model is assumed to be enabled (see Chapter 18: Modeling Partially Premixed Combustion in the separate User's Guide), so that the unburned density and laminar flame speed are available as polynomials. See Chapter 3: Additional Macros for Writing UDFs for details on the `NULLP`, `THREAD_STORAGE`, and `SV_VARS` macros.

```
/**********************************************************************
    UDF that specifies a custom turbulent flame speed and source
    for the premixed combustion model
**********************************************************************/

#include "udf.h"
#include "sg_pdf.h" /* not included in udf.h so must include here  */

DEFINE_TURB_PREMIX_SOURCE(turb_flame_src,c,t,turb_flame_speed,source)
{
  real up = TRB_VEL_SCAL(c,t);
  real ut, ul, grad_c, rho_u, Xl, DV[ND_ND];

  ul = C_LAM_FLAME_SPEED(c,t);
  Calculate_unburnt_rho_and_Xl(t, &rho_u, &Xl);

  if( NNULLP(THREAD_STORAGE(t,SV_PREMIXC_G)) )
    {
      NV_V(DV, =, C_STORAGE_R_NV(c,t,SV_PREMIXC_G));
      grad_c = sqrt(NV_DOT(DV,DV) );
    }

  ut = ul*sqrt( 1. + SQR(up/ul) );

  *turb_flame_speed = ut;
  *source = rho_u*ut*grad_c;
}
```

Hooking a Turbulent Premixed Source UDF to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_TURB_PREMIX_SOURCE` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first `DEFINE` macro argument (e.g., `turb_flame_src`) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.24: Hooking `DEFINE_TURB_PREMIX_SOURCE` UDFs for details.

2.3.25 `DEFINE_TURB_SCHMIDT` **UDF**

The `DEFINE_TURB_SCHMIDT` macro can be used to specify the turbulent Schmidt numbers of all transported species in ANSYS FLUENT, for single-phase flows.

Description

The turbulent Schmidt number, denoted Sc_t, controls the turbulent diffusion of species transported in ANSYS FLUENT. You can use `DEFINE_TURB_SCHMIDT` to specify Sc_t for each species solved.

Usage

`DEFINE_TURB_SCHMIDT(name,c,t, i)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index of cell on which the Turbulent Schmidt number function is to be applied.
`Thread *t`	Pointer to cell thread.
`int i`	Species index.

Function returns
`real`

There are four arguments to `DEFINE_TURB_SCHMIDT`: `name`, `c`, `t` and `i`. You supply `name`, the name of the UDF. `c`, `t` and `i` are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the `real` value for the turbulent Schmidt number to the solver.

Example

The following example sets Sc_t to be inversely proportional to the species index. Hence, the first species in the materials list will have the smallest turbulent diffusion, and the last species will have the largest turbulent diffusion.

```
#include "udf.h"

DEFINE_TURB_SCHMIDT(udf_sct, c, t, i)
{
  return 1./((real)i+1.);
}
```

Hooking a Turbulent Schmidt Number UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_TURB_SCHMIDT is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., udf_sct in the above example) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.25: Hooking DEFINE_TURB_SCHMIDT UDFs for details.

2.3.26   DEFINE_TURBULENT_VISCOSITY

Description

You can use DEFINE_TURBULENT_VISCOSITY to specify a custom turbulent viscosity function for the Spalart-Allmaras, k-ϵ, and k-ω turbulence models for single-phase applications. In addition, for 3d versions of ANSYS FLUENT you can specify a subgrid-scale turbulent viscosity UDF for the large eddy simulation model. For Eulerian multiphase flows, turbulent viscosity UDFs can be assigned on a per-phase basis, and/or to the mixture, depending on the turbulence model. See Table 2.3.6 for details.

Table 2.3.6: Eulerian Multiphase Model and DEFINE_TURBULENT_VISCOSITY
UDF Usage

Turbulence Model	Phase that Turbulent Viscosity UDF Is Specified On
k-ϵ **Mixture**	mixture, primary and secondary phases
k-ϵ **Dispersed**	primary and secondary phases
k-ϵ **Per-Phase**	primary and secondary phases

Usage

DEFINE_TURBULENT_VISCOSITY(name,c,t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the turbulent viscosity is to be applied.

Function returns
real

There are three arguments to DEFINE_TURBULENT_VISCOSITY: name, c, and t. You supply
name, the name of the UDF. c and t are variables that are passed by the ANSYS FLUENT
solver to your UDF. Your UDF will need to return the real value of the turbulent
viscosity to the solver.

Example 1 - Single Phase Turbulent Viscosity UDF

The following UDF, named user_mu_t, defines a custom turbulent viscosity for the stan-
dard k-ϵ turbulence model. Note that the value of M_keCmu in the example is defined
through the graphical user interface, but made accessible to all UDFs. The source code
can be interpreted or compiled in ANSYS FLUENT.

```
/**********************************************************************
    UDF that specifies a custom turbulent viscosity for standard
    k-epsilon formulation
**********************************************************************/

#include "udf.h"

DEFINE_TURBULENT_VISCOSITY(user_mu_t,c,t)
{
  real mu_t;
  real rho = C_R(c,t);
  real k   = C_K(c,t);
  real d   = C_D(c,t);

  mu_t = M_keCmu*rho*SQR(k)/d;

  return mu_t;
}
```

Example 2 - Multiphase Turbulent Viscosity UDF

```
/**********************************************************************
    Custom turbulent viscosity functions for each phase and the
    mixture in a two-phase multiphase flow
**********************************************************************/
#include "udf.h"

DEFINE_TURBULENT_VISCOSITY(mu_t_ke_mixture, c, t)
{
  real mu_t;
  real rho = C_R(c,t);
  real k   = C_K(c,t);
  real d   = C_D(c,t);
  real cmu = M_keCmu;

  mu_t = rho*cmu*k*k/d;

  return mu_t;
}

DEFINE_TURBULENT_VISCOSITY(mu_t_ke_1, c, t)
{
  Thread *tm = lookup_thread_by_id(DOMAIN_SUPER_DOMAIN(THREAD_DOMAIN(t)),
```

```
                                              t->id);
  CACHE_T_SV_R (density,   t,   SV_DENSITY);
  CACHE_T_SV_R (mu_t,      t,   SV_MU_T);
  CACHE_T_SV_R (density_m, tm,  SV_DENSITY);
  CACHE_T_SV_R (mu_t_m,    tm,  SV_MU_T);

  return density[c]/density_m[c]*mu_t_m[c];
}


DEFINE_TURBULENT_VISCOSITY(mu_t_ke_2, c, t)
{
  Thread *tm = lookup_thread_by_id(DOMAIN_SUPER_DOMAIN(THREAD_DOMAIN(t)),
                                   t->id);
  CACHE_T_SV_R (density,   t,   SV_DENSITY);
  CACHE_T_SV_R (mu_t,      t,   SV_MU_T);
  CACHE_T_SV_R (density_m, tm,  SV_DENSITY);
  CACHE_T_SV_R (mu_t_m,    tm,  SV_MU_T);

  return density[c]/density_m[c]*mu_t_m[c];
}
```

Hooking a Turbulent Viscosity UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_TURBULENT_VISCOSITY is interpreted
(Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the function
name(s) that you specified in the DEFINE macro argument(s) (for example user_mu_t
for single phase, or mu_t_ke_mixture, mu_t_ke_1, and mu_t_ke_2 for multiphase) will
become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See
Section 6.2.26: Hooking DEFINE_TURBULENT_VISCOSITY UDFs for details.

2.3.27 DEFINE_VR_RATE

Description

You can use DEFINE_VR_RATE to specify a custom volumetric reaction rate for a single reaction or for multiple reactions. During ANSYS FLUENT execution, DEFINE_VR_RATE is called for every reaction in every single cell. A DEFINE_VR_RATE UDF is compatible with the laminar finite-rate model, but you must make sure that the stiff chemistry option is disabled.

Usage

DEFINE_VR_RATE(name,c,t,r,mw,yi,rr,rr_t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the volumetric reaction rate is to be applied.
Reaction *r	Pointer to data structure that represents the current reaction.
real *mw	Pointer to array of species molecular weights.
real *yi	Pointer to array of the species mass fractions.
real *rr	Pointer to laminar reaction rate.
real *rr_t	Pointer to turbulent reaction rate.

Function returns
void

There are eight arguments to DEFINE_VR_RATE: name, c, t, r, mw, yi, rr, and rr_t. You supply name, the name of the UDF. c, t, r, mw, yi, rr, and rr_t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the values referenced by the **real** pointers rr and rr_t to the laminar and turbulent reaction rates, respectively.

rr and rr_t (defined by the UDF) are computed and the lower of the two values is used when the finite-rate/eddy-dissipation chemical reaction mechanism used. Note that rr and rr_t are conversion rates in kmol/m^3-s. These rates, when multiplied by the respective stoichiometric coefficients, yield the production/consumption rates of the individual chemical components.

Example 1

The following UDF, named vol_reac_rate, specifies a volume reaction rate. The function must be executed as a compiled UDF in ANSYS FLUENT.

```
/************************************************************************
    UDF for specifying a volume reaction rate
    The basics of ANSYS FLUENT's calculation of reaction rates:  only an
    Arrhenius ("finite rate") reaction rate is calculated
    from the inputs given by the user in the graphical user interface
*************************************************************************/

#include "udf.h"

DEFINE_VR_RATE(vol_reac_rate,c,t,r,wk,yk,rate,rr_t)
{
  real ci, prod;
  int i;

/* Calculate Arrhenius reaction rate  */

  prod = 1.;

  for(i = 0; i < r->n_reactants; i++)
    {
       ci    = C_R(c,t) * yk[r->reactant[i]] / wk[r->reactant[i]];
       prod  *= pow(ci, r->exp_reactant[i]);
    }
  *rate = r->A * exp( - r->E / (UNIVERSAL_GAS_CONSTANT * C_T(c,t))) *
                                    pow(C_T(c,t), r->b) * prod;

  *rr_t = *rate;

  /* No "return..;" value. */
}
```

Example 2

When multiple reactions are specified, a volume reaction rate UDF is called several times in each cell. Different values are assigned to the pointer `r`, depending on which reaction the UDF is being called for. Therefore, you will need to determine which reaction is being called, and return the correct rates for that reaction. Reactions can be identified by their name through the `r->name` statement. To test whether a given reaction has the name `reaction-1`, for example, you can use the following C construct:

```
if (!strcmp(r->name, "reaction-1"))
    {
      ....  /* r->name is identical to "reaction-1" ... */
    }
```

i Note that `strcmp(r->name, "reaction-1")` returns 0 which is equal to FALSE when the two strings are identical.

It should be noted that `DEFINE_VR_RATE` defines only the reaction rate for a predefined stoichiometric equation (set in the Reactions dialog box) thus providing an alternative to the Arrhenius rate model. `DEFINE_VR_RATE` does not directly address the particular rate of species creation or depletion; this is done by the ANSYS FLUENT solver using the reaction rate supplied by your UDF.

The following is a source code template that shows how to use `DEFINE_VR_RATE` in connection with more than one user-specified reaction. Note that ANSYS FLUENT always calculates the `rr` and `rr_t` reaction rates before the UDF is called. Consequently, the values that are calculated are available only in the given variables when the UDF is called.

```
/***********************************************************************
   Multiple reaction UDF that specifies different reaction rates
   for different volumetric chemical reactions
***********************************************************************/
#include "udf.h"

DEFINE_VR_RATE(myrate,c,t,r,mw,yi,rr,rr_t)
{
 /*If more than one reaction is defined, it is necessary to distinguish
   between these using the names of the reactions.                    */

      if (!strcmp(r->name, "reaction-1"))
        {
          /* Reaction 1 */
        }
      else if (!strcmp(r->name, "reaction-2"))
        {
          /* Reaction 2 */
        }
      else
        {
/*        Message("Unknown Reaction\n"); */
        }
/*    Message("Actual Reaction: %s\n",r->name); */

}
```

Hooking a Volumetric Reaction Rate UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_VR_RATE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., myrate) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.2.27: Hooking DEFINE_VR_RATE UDFs for details.

2.3.28 DEFINE_WALL_FUNCTIONS

Description

You can use DEFINE_WALL_FUNCTIONS to provide custom wall functions for applications when you want to replace the standard wall functions in ANSYS FLUENT. Note that this is available only for use with the k-ϵ turbulence models.

Usage

DEFINE_WALL_FUNCTIONS(name,f,t,c0,t0,wf_ret,yPlus,Emod)

Argument Type	Description
symbol name	UDF name.
face_t f	face index.
Thread *t	pointer to cell thread
cell_t c0	cell index.
Thread *t0	pointer to face thread.
int wf_ret	wall function index
real yPlus	y+ value
real Emod	wall function E constant

Function returns
real

There are eight arguments to DEFINE_WALL_FUNCTIONS: name, f, t, c0, t0, wf_ret, yPlus, and Emod. You supply name, the name of the UDF. f, t, c0, t0, wf_ret, yPlus, and Emod are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the real value of the wall functions U+, dU+, and dY+ for laminar and turbulent regions and return them to the solver.

Example

The following UDF, named user_log_law, computes U+ and dU+, and dY+ for laminar and turbulent regions using DEFINE_WALL_FUNCTIONS. The source code can be interpreted or compiled in ANSYS FLUENT.

```
/********************************************************************
User-defined wall functions: separated into turbulent and
laminar regimes
********************************************************************/
#include "udf.h"

DEFINE_WALL_FUNCTIONS(user_log_law, f, t, c0, t0, wf_ret, yPlus, Emod)
{
  real wf_value;

  switch (wf_ret)
    {
    case UPLUS_LAM:
      wf_value = yPlus;
      break;
    case UPLUS_TRB:
      wf_value = log(Emod*yPlus)/KAPPA;
      break;
    case DUPLUS_LAM:
      wf_value = 1.0;
      break;
    case DUPLUS_TRB:
      wf_value = 1./(KAPPA*yPlus);
      break;
    case D2UPLUS_TRB:
      wf_value = -1./(KAPPA*yPlus*yPlus);
      break;
    default:
printf("Wall function return value unavailable\n");
    }
  return wf_value;
}
```

Hooking a Wall Function UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_WALL_FUNCTIONS is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_log_law) will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. See Section 6.2.28: Hooking DEFINE_WALL_FUNCTIONS UDFs for details.

2.4 Multiphase DEFINE Macros

The DEFINE macros presented in this section are used for multiphase applications, *only*.

Table 2.4.1 provides a quick reference guide to the multiphase-specific DEFINE macros, the functions they are used to define, and the dialog boxes in which they are activated in ANSYS FLUENT. Definitions of each DEFINE macro are listed in the udf.h header file (see Appendix C).

Appendix B contains a list of general purpose DEFINE macros that can also be used to define UDFs for multiphase cases. For example, the general purpose DEFINE_PROPERTY macro is used to define a surface tension coefficient UDF for the multiphase VOF model. See Section 2.3.16: DEFINE_PROPERTY UDFs for details.

- Section 2.4.1: DEFINE_CAVITATION_RATE

- Section 2.4.2: DEFINE_EXCHANGE_PROPERTY

- Section 2.4.3: DEFINE_HET_RXN_RATE

- Section 2.4.4: DEFINE_MASS_TRANSFER

- Section 2.4.5: DEFINE_VECTOR_EXCHANGE_PROPERTY

Table 2.4.1: Quick Reference Guide for Multiphase DEFINE Macros

Model	Function	DEFINE Macro	Dialog Box Activated
VOF	mass transfer	DEFINE_MASS_TRANSFER	Phase Interaction
	heterogeneous reaction rate	DEFINE_HET_RXN_RATE	Phase Interaction
Mixture	mass transfer	DEFINE_MASS_TRANSFER	Phase Interaction
	drag coefficient	DEFINE_EXCHANGE_PROPERTY	Phase Interaction
	slip velocity	DEFINE_VECTOR_EXCHANGE_ _PROPERTY	Phase Interaction
	cavitation rate	DEFINE_CAVITATION_RATE	User-Defined Function Hooks
	heterogeneous reaction rate	DEFINE_HET_RXN_RATE	Phase Interaction
Eulerian	mass transfer	DEFINE_MASS_TRANSFER	Phase Interaction
	heat transfer	DEFINE_EXCHANGE_PROPERTY	Phase Interaction
	drag coefficient	DEFINE_EXCHANGE_PROPERTY	Phase Interaction
	lift coefficient	DEFINE_EXCHANGE_PROPERTY	Phase Interaction
	heterogeneous reaction rate	DEFINE_HET_RXN_RATE	Phase Interaction

2.4.1 DEFINE_CAVITATION_RATE

Description

You can use DEFINE_CAVITATION_RATE to model the cavitation source terms R_e and R_c in the vapor mass fraction transport equation used in the Singhal et. al model (see Equation 16.7-24 in the separate Theory Guide). Assuming m_{dot} denotes the mass-transfer rate between liquid and vapor phases, we have

$$R_e = MAX[m_{dot}, 0]f_1$$

$$R_c = MAX[-m_{dot}, 0]f_v$$

where f_1 and f_v are the mass-fraction of the liquid and vapor phase, respectively.

DEFINE_CAVITATION_RATE is used to calculate m_{dot} only. The values of R_e and R_c are computed by the solver, accordingly.

Usage

DEFINE_CAVITATION_RATE(name,c,t,p,rhoV,rhoL,mafV,p_v,cigma,f_gas,m_dot)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to the mixture-level thread.
real *p[c]	Pointer to shared pressure.
real *rhoV[c]	Pointer to vapor density.
real *rhoL[c]	Pointer to liquid density.
real *mafV[c]	Pointer to vapor mass fraction.
real *p_v	Pointer to vaporization pressure.
real *cigma	Pointer to liquid surface tension coefficient.
real *f_gas	Pointer to the prescribed mass fraction of non condensable gases.
real *m_dot	Pointer to cavitation mass transfer rate.

Function returns
void

There are eleven arguments to DEFINE_CAVITATION_RATE: name, c, t, p, rhoV, rhoL, mafV, p_v, cigma, f_gas, and m_dot. You supply name, the name of the UDF. c, t, p, rhoV, rhoL, mafV, p_v, cigma, f_gas, and m_dot are variables that are passed by the ANSYS

FLUENT solver to your UDF. Your UDF will need to set the value referenced by the `real` pointer `m_dot` to the cavitation rate.

Example

The following UDF named `c_rate`, is an example of a cavitation model for a multiphase mixture that is different from the default model in ANSYS FLUENT. This cavitation model calculates the cavitation mass transfer rates between the liquid and vapor phase depending on fluid pressure (`*p`), turbulence kinetic energy (`C_K(c,t)`), and the liquid vaporization pressure (`*p_v`).

In general, the existence of turbulence enhances cavitation. In this example, the turbulence effect is taken into account by increasing the cavitation pressure by `0.195*C_R(c,t) * C_K(c,t)`. The pressure `p_vapor` that determines whether cavitation occurs increases from `p_v` to

```
p_v + 0.195 * C_R(c,t) * C_K(c,t)
```

When the absolute fluid pressure (`ABS_P`) is lower than `p_vapor`, then liquid evaporates to vapor (R_e). When it is greater than `p_vapor`, vapor condenses to liquid (R_c).

The evaporation rate is calculated by

```
  If ABS_P < p_vapor, then
    c_evap * rhoV[c] * sqrt(2.0/3.0*rhoL[c]) * ABS(p_vapor - ABS_P(p[c]))
```

The condensation rate is

```
  If ABS_P > p_vapor, then
    -c_con*rhoL[c] * sqrt(2.0/3.0*rhoL[c]) * ABS(p_vapor - ABS_P(p[c]))
```

where `c_evap` and `c_con` are model coefficients.

```
/************************************************************************
   UDF that is an example of a cavitation model different from default.
   Can be interpreted or compiled.
************************************************************************/

#include "udf.h"

#define c_evap 1.0
#define c_con 0.1

DEFINE_CAVITATION_RATE(c_rate,c,t,p,rhoV,rhoL,mafV,p_v,cigma,f_gas, m_dot)
{
    real p_vapor = *p_v;
    real dp, dp0, source;
    p_vapor  += MIN(0.195*C_R(c,t)*C_K(c,t), 5.0*p_vapor);
    dp = p_vapor - ABS_P(p[c], op_pres);
    dp0 = MAX(0.1, ABS(dp));
    source = sqrt(2.0/3.0*rhoL[c])*dp0;

    if(dp > 0.0)
      *m_dot = c_evap*rhoV[c]*source;
    else
      *m_dot = -c_con*rhoL[c]*source;
}
```

Hooking a Cavitation Rate UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_CAVITATION_RATE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., c_rate) will become visible and selectable in the User-Defined Function Hooks dialog box in ANSYS FLUENT. See Section 6.3.1: Hooking DEFINE_CAVITATION_RATE UDFs for details.

2.4.2   DEFINE_EXCHANGE_PROPERTY

Description

You can use DEFINE_EXCHANGE_PROPERTY to specify UDFs for some phase interaction variables in multiphase models. These include net heat transfer rates between phases, and drag and lift coefficient functions. Below is a list of user-defined functions that can be specified using DEFINE_EXCHANGE_PROPERTY for the multiphase models in ANSYS FLUENT. Note that there are some phase interaction variables such as vaporization pressure and surface tension coefficient (cavitation parameters) that are defined using DEFINE_PROPERTY. See Section 2.3.16: DEFINE_PROPERTY UDFs for details.

Table 2.4.2: DEFINE_EXCHANGE_PROPERTY Variables

Mixture Model	Eulerian Model
drag exchange coefficient	net heat transfer rate drag coefficient lift coefficient

Usage

DEFINE_EXCHANGE_PROPERTY(name,c,mixture_thread,second_column_phase_index, first_column_phase_index)

i Note that all of the arguments to a DEFINE macro must be placed on the same line in your source code. Splitting the DEFINE statement onto several lines will result in a compilation error.

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *mixture_thread	Pointer to the mixture-level thread.
int second_column_phase_index	Identifier that corresponds to the pair of phases in your multiphase flow that you are specifying a slip velocity for. The identifiers correspond to the phases you select in the Phase Interaction dialog box in the graphical user interface. An index of 0 corresponds to the primary phase, and is incremented by one for each secondary phase.
int first_column_phase_index	See int second_column_phase_index.

Function returns
real

There are five arguments to DEFINE␣EXCHANGE␣PROPERTY: name, c, mixture␣thread, second␣column␣phase␣index, and first␣column␣phase␣index. You supply name, the name of the UDF. c, mixture␣thread, second␣column␣phase␣index, and first␣column␣phase␣index are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value of the lift coefficient, drag exchange coefficient, heat or mass transfer to the solver.

Example 1 - Custom Drag Law

The following UDF, named custom␣drag, can be used to customize the default Syamlal drag law in ANSYS FLUENT. The default drag law uses 0.8 (for void <=0.85) and 2.65 (void >0.85) for bfac. This results in a minimum fluid velocity of 25 cm/s. The UDF modifies the drag law to result in a minimum fluid velocity of 8 cm/s, using 0.28 and 9.07 for the bfac parameters.

```
/*********************************************************************
   UDF for customizing the default Syamlal drag law in ANSYS FLUENT
 *********************************************************************/

#include "udf.h"

#define pi 4.*atan(1.)
#define diam2 3.e-4

DEFINE_EXCHANGE_PROPERTY(custom_drag,cell,mix_thread,s_col,f_col)
{
 Thread *thread_g, *thread_s;
 real x_vel_g, x_vel_s, y_vel_g, y_vel_s, abs_v, slip_x, slip_y,
      rho_g, rho_s, mu_g, reyp, afac,
      bfac, void_g, vfac, fdrgs, taup, k_g_s;

 /* find the threads for the gas (primary) */
 /* and solids (secondary phases)          */

 thread_g = THREAD_SUB_THREAD(mix_thread, s_col);/* gas phase  */
 thread_s = THREAD_SUB_THREAD(mix_thread, f_col);/* solid phase*/

 /* find phase velocities and properties*/

 x_vel_g = C_U(cell, thread_g);
 y_vel_g = C_V(cell, thread_g);

 x_vel_s = C_U(cell, thread_s);
```

```
    y_vel_s = C_V(cell, thread_s);

    slip_x = x_vel_g - x_vel_s;
    slip_y = y_vel_g - y_vel_s;

    rho_g = C_R(cell, thread_g);
    rho_s = C_R(cell, thread_s);

    mu_g = C_MU_L(cell, thread_g);

    /*compute slip*/
    abs_v = sqrt(slip_x*slip_x + slip_y*slip_y);

    /*compute Reynold's number*/

    reyp = rho_g*abs_v*diam2/mu_g;

    /* compute particle relaxation time */

    taup = rho_s*diam2*diam2/18./mu_g;

    void_g = C_VOF(cell, thread_g);/* gas vol frac*/

    /*compute drag and return drag coeff, k_g_s*/

    afac = pow(void_g,4.14);

    if(void_g<=0.85)
     bfac = 0.281632*pow(void_g, 1.28);
    else
     bfac = pow(void_g, 9.076960);

    vfac = 0.5*(afac-0.06*reyp+sqrt(0.0036*reyp*reyp+0.12*reyp*(2.*bfac-
                 afac)+afac*afac));
    fdrgs = void_g*(pow((0.63*sqrt(reyp)/
                              vfac+4.8*sqrt(vfac)/vfac),2))/24.0;

    k_g_s = (1.-void_g)*rho_s*fdrgs/taup;

    return k_g_s;
    }
```

Example 2 - Heat Transfer

The following UDF, named `heat_udf`, specifies a coefficient that when multiplied by the temperature difference between the dispersed and continuous phases, is equal to the net rate of heat transfer per unit volume.

```
#include "udf.h"

#define PR_NUMBER(cp,mu,k) ((cp)*(mu)/(k))
#define IP_HEAT_COEFF(vof,k,nu,d) ((vof)*6.*(k)*(Nu)/(d)/(d))

static real
heat_ranz_marshall(cell_t c, Thread *ti, Thread *tj)
{
 real h;
 real d = C_PHASE_DIAMETER(c,tj);
 real k = C_K_L(c,ti);
 real NV_VEC(v), vel, Re, Pr, Nu;

 NV_DD(v,=,C_U(c,tj),C_V(c,tj),C_W(c,tj),-,C_U(c,ti),C_V(c,ti),C_W(c,ti));
 vel = NV_MAG(v);

 Re = RE_NUMBER(C_R(c,ti),vel,d,C_MU_L(c,ti));
 Pr = PR_NUMBER (C_CP(c,ti),C_MU_L(c,ti),k);
 Nu = 2. + 0.6*sqrt(Re)*pow(Pr,1./3.);

 h = IP_HEAT_COEFF(C_VOF(c,tj),k,Nu,d);
return h;
}

DEFINE_EXCHANGE_PROPERTY(heat_udf, c, t, i, j)
{
  Thread *ti = THREAD_SUB_THREAD(t,i);
  Thread *tj = THREAD_SUB_THREAD(t,j);
  real val;

  val = heat_ranz_marshall(c,ti, tj);
  return val;
}
```

Hooking an Exchange Property UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_EXCHANGE_PROPERTY is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., heat_udf) will become visible and selectable in the Phase Interaction dialog box in ANSYS FLUENT. See Section 6.3.2: Hooking DEFINE_EXCHANGE_PROPERTY UDFs for details.

2.4.3 DEFINE_HET_RXN_RATE

Description

You need to use DEFINE_HET_RXN_RATE to specify reaction rates for heterogeneous reactions. A heterogeneous reaction is one that involves reactants and products from more than one phase. Unlike DEFINE_VR_RATE, a DEFINE_HET_RXN_RATE UDF can be specified differently for different heterogeneous reactions.

During ANSYS FLUENT execution, the DEFINE_HET_RXN_RATE UDF for each heterogeneous reaction that is defined is called in every fluid cell. ANSYS FLUENT will use the reaction rate specified by the UDF to compute production/destruction of the species participating in the reaction, as well as heat and momentum transfer across phases due to the reaction.

A heterogeneous reaction is typically used to define reactions involving species of different phases. The bulk phase can participate in the reaction if the phase does not have any species (i.e. phase has fluid material instead of mixture material). Heterogeneous reactions are defined in the Phase Interaction dialog box.

Usage

DEFINE_HET_RXN_RATE(name,c,t,r,mw,yi,rr,rr_t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Cell thread (mixture level) on which heterogeneous reaction rate is to be applied.
Hetero_Reaction *r	Pointer to data structure that represents the current heterogeneous reaction (see sg_mphase.h).
real mw[MAX_PHASES][MAX_SPE_EQNS]	Matrix of species molecular weights. mw[i][j] will give molecular weight of species with ID j in phase with index i. For phase which has fluid material, the molecular weight can be accessed as mw[i][0].
real yi[MAX_PHASES][MAX_SPE_EQNS]	Matrix of species mass fractions. yi[i][j] will give mass fraction of species with ID j in phase with index i. For phase which has fluid material, yi[i][0] will be 1.
real *rr	Pointer to laminar reaction rate.
real *rr_t	Currently not used. Provided for future use.

Function returns
void

There are eight arguments to DEFINE_HET_RXN_RATE: name, c, t, r, mw, yi, rr, and rr_t. You supply name, the name of the UDF. c, t, r, mw, yi, rr, and rr_t are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the values referenced by the real pointer rr. The values must be specified in $\frac{kmol\,m^3}{s}$ (where the volume is the cell volume).

Example

The following compiled UDF named user_evap_condens_react defines the reaction rate required to simulate evaporation or condensation on the surface of droplets. Such a reaction can be formally described by the following:

$$H_2O_{(liq)} \longleftarrow \longrightarrow H_2O_{(gas)} \tag{2.4-1}$$

Here, gas is a primary phase mixture of two species: $H_2O_{(gas)}$ and air. Droplets constitute the secondary phase and represent a mixture of one species - $H_2O_{(liq)}$. Single-species mixtures are allowed in multiphase models.

The formulation for the reaction rate follows the model for particle evaporation that is defined in Section 15.4.2: Droplet Vaporization (Law 2) in the separate Theory Guide.

```
#include "udf.h"

/*Constants used in psat_h2o to calculate saturation pressure*/

#define PSAT_A 0.01
#define PSAT_TP 338.15
#define C_LOOP 8
#define H2O_PC 22.089E6
#define H2O_TC 647.286

/*user inputs*/

#define MAX_SPE_EQNS_PRIM 2 /*total number of species in primary phase*/
#define  index_evap_primary 0  /*evaporating species index in primary phase*/
#define  prim_index 0  /*index of primary phase*/
#define  P_OPER 101325    /*operating pressure equal to GUI value*/

/*end of user inputs*/

/*************************************************************/
/* UDF for specifying an interfacial area density           */
/*************************************************************/
double psat_h2o(double tsat)
/*                                                           */
/* Computes saturation pressure of water vapor              */
/* as function of temperature                                */
/* Equation is taken from THERMODYNAMIC PROPERTIES IN SI,    */
/* by Reynolds, 1979                                         */
/* Returns pressure in PASCALS, given temperature in KELVIN  */
{
   int i;
   double var1,sum1,ans1,psat;
   double constants[8]={-7.4192420, 2.97221E-1, -1.155286E-1,
    8.68563E-3, 1.094098E-3, -4.39993E-3, 2.520658E-3, -5.218684E-4};

/* var1 is an expression that is used in the summation loop */
   var1 = PSAT_A*(tsat-PSAT_TP);
```

```
/* Compute summation loop */
   i = 0;
   sum1 = 0.0;
   while (i < C_LOOP){
       sum1+=constants[i]*pow(var1,i);
++i;
   }
ans1 = sum1*(H2O_TC/tsat-1.0);

/* compute exponential to determine result */
/* psat has units of Pascals              */

psat = H2O_PC*exp(ans1);
return psat;
}


DEFINE_HET_RXN_RATE(user_evap_condens_react, c, t, hr, mw, yi, rr, rr_t)
{
    Thread **pt = THREAD_SUB_THREADS(t);
    Thread *tp = pt[0];
    Thread *ts = pt[1];
    int i;
    real concentration_evap_primary, accum = 0., mole_frac_evap_prim,
        concentration_sat ;
    real T_prim = C_T(c,tp); /*primary phase (gas) temperature*/
    real T_sec = C_T(c,ts);  /*secondary phase (droplet) temperature*/
    real diam = C_PHASE_DIAMETER(c,ts);  /*secondary phase diameter*/
    real D_evap_prim = C_DIFF_EFF(c,tp,index_evap_primary)
      - 0.7*C_MU_T(c,tp)/C_R(c,tp);
       /*primary phase species turbulent diffusivity*/
    real Re, Sc, Nu, urel, urelx,urely,urelz=0., mass_coeff, area_density,
        flux_evap ;

    if(Data_Valid_P())
    {
      urelx = C_U(c,tp) - C_U(c,ts);
      urely = C_V(c,tp) - C_V(c,ts);

      #if RP_3D
        urelz = C_W(c,tp) - C_W(c,ts);
      #endif
```

```
        urel = sqrt(urelx*urelx + urely*urely + urelz*urelz);
          /*relative velocity*/

        Re = urel * diam * C_R(c,tp) / C_MU_L(c,tp);

        Sc = C_MU_L(c,tp) / C_R(c,tp) / D_evap_prim ;

        Nu =  2. + 0.6 * pow(Re, 0.5)* pow(Sc, 0.333);

        mass_coeff = Nu * D_evap_prim / diam ;

        for (i=0; i < MAX_SPE_EQNS_PRIM ; i++)
          {
            accum = accum + C_YI(c,tp,i)/mw[i][prim_index];
          }

        mole_frac_evap_prim = C_YI(c,tp,index_evap_primary )
                     / mw[index_evap_primary][prim_index] / accum;

        concentration_evap_primary = mole_frac_evap_prim * P_OPER
                     / UNIVERSAL_GAS_CONSTANT / T_prim ;

        concentration_sat = psat_h2o(T_sec)/UNIVERSAL_GAS_CONSTANT/T_sec ;

        area_density = 6. * C_VOF(c,ts) / diam ;

        flux_evap = mass_coeff *
             (concentration_sat - concentration_evap_primary ) ;

        *rr = area_density * flux_evap ;
        }
}
```

Hooking a Heterogeneous Reaction Rate UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_HET_RXN_RATE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., user_evap_condens_react) will become visible and selectable under Reaction Rate Function in the Reactions tab of the Phase Interaction dialog box. (Note you will first need to specify the Total Number of Reactions greater than 0.) See Section 6.3.3: Hooking DEFINE_HET_RXN_RATE UDFs for details.

2.4.4 DEFINE_MASS_TRANSFER

Description

You can use DEFINE_MASS_TRANSFER when you want to model mass transfer in a multi-phase problem. The mass transfer rate specified using a DEFINE_MASS_TRANSFER UDF is used to compute mass, momentum, energy, and species sources for the phases involved in the mass transfer. For problems in which species transport is enabled, the mass transfer will be from one species in one phase, to another species in another phase. If one of the phases does not have a mixture material associated with it, then the mass transfer will be with the bulk fluid of that phase.

Usage

DEFINE_MASS_TRANSFER(name,c,mixture_thread,from_phase_index, from_species_index, to_phase_index,to_species_index)

i Note that all of the arguments to a DEFINE macro need to be placed on the same line in your source code. Splitting the DEFINE statement onto several lines will result in a compilation error.

Argument Type	Description
symbol name	UDF name.
cell_t c	Index of cell on the thread pointed to by mixture_thread.
Thread *mixture_thread	Pointer to mixture-level thread.
int from_phase_index	Index of phase from which mass is transferred.
int from_species_index	ID of species from which mass is transferred (ID= -1 if phase does not have mixture material).
int to_phase_index	Index of phase to which mass is transferred.
int to_species_index	ID of species to which mass is transferred (ID= -1 if phase does not have mixture material).

Function returns: real

There are seven arguments to DEFINE_MASS_TRANSFER: name, c, mixture_thread, from_phase_index, from_species_index, to_phase_index, to_species_index. You supply name, the name of the UDF. The variables c, mixture_thread, from_phase_index, from_species_index, to_phase_index, and to_species_index are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to return the real value of the mass transfer to the solver in the units of kg/m^3/s.

> *i* The arguments from_species_index and to_species_index are relevant
> for multiphase species transport problems only, and only if the respective
> phase has a mixture material associated with it.

Example

The following UDF, named liq_gas_source, specifies a simple mass transfer coefficient
based on saturation temperature:

> *i* Note that in the example that follows, the DEFINE_MASS_TRANSFER state-
> ment is broken up into two lines for the sake of readability. In your source
> file, you must make sure that the DEFINE statement is on one line only.

```
/* UDF to define a simple mass transfer based on Saturation
   Temperature. The "from" phase is the gas and the "to" phase is the
   liquid phase  */

#include "udf.h"

DEFINE_MASS_TRANSFER(liq_gas_source, cell, thread, from_index,
from_species_index, to_index, to_species_index)
{
   real m_lg;
   real T_SAT = 373.15;
   Thread *gas = THREAD_SUB_THREAD(thread, from_index);
   Thread *liq = THREAD_SUB_THREAD(thread, to_index);

   m_lg = 0.;
   if (C_T(cell, liq) >= T_SAT)
     {
       m_lg = -0.1*C_VOF(cell,liq)*C_R(cell,liq)*
              fabs(C_T(cell,liq)-T_SAT)/T_SAT;
     }
   if ((m_lg == 0. ) && (C_T(cell, gas) <= T_SAT))
     {
       m_lg = 0.1*C_VOF(cell,gas)*C_R(cell,gas)*
         fabs(T_SAT-C_T(cell,gas))/T_SAT;
     }

   return (m_lg);
}
```

Hooking a Mass Transfer **UDF to** ANSYS FLUENT

After the UDF that you have defined using `DEFINE_MASS_TRANSFER` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first `DEFINE` macro argument (e.g., `liq_gas_source`) will become visible and selectable under Mass Transfer when you select the Mass tab option in the Phase Interaction dialog box and specify the Number of Mass Transfer Functions. See Section 6.3.4: Hooking `DEFINE_MASS_TRANSFER` UDFs for details.

2.4.5 `DEFINE_VECTOR_EXCHANGE_PROPERTY`

Description

You can use `DEFINE_VECTOR_EXCHANGE_PROPERTY` to specify custom slip velocities for the multiphase Mixture model.

Usage

`DEFINE_VECTOR_EXCHANGE_PROPERTY(name,c,mixture_thread,`
`second_column_phase_index,first_column_phase_index,vector_result)`

i Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *mixture_thread`	Pointer to cell thread of mixture domain.
`int second_column_phase_index`	Index of second phase in phase interaction.
`int first_column_phase_index`	Index of first phase in phase interaction.
`real *vector_result`	Pointer to slip velocity vector.

Function returns: `void`

There are six arguments to `DEFINE_VECTOR_EXCHANGE_PROPERTY`: `name`, `c`, `mixture_thread`, `second_column_phase_index`, `first_column_phase_index`, and `vector_result`. You supply `name`, the name of the UDF. `c`, `mixture_thread`, `second_column_phase_index`, `first_column_phase_index`, and `vector_result` are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the values referenced by the `real` pointer to the slip velocity vector (`vector_result`) to the components of the slip velocity vector (e.g., `vector_result[0]`, `vector_result[1]` for a 2D problem).

Example

The following UDF, named `custom_slip`, specifies a custom slip velocity in a two-phase mixture problem.

> *i* Note that in the example that follows, the `DEFINE_VECTOR_EXCHANGE_PROPERTY` statement is broken up into two lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/***************************************************************
   UDF for a defining a custom slip velocity in a 2-phase
   mixture problem
***************************************************************/

#include "udf.h"

DEFINE_VECTOR_EXCHANGE_PROPERTY(custom_slip,c,mixture_thread,
second_column_phase_index,first_column_phase_index,vector_result)
{
 real grav[2] = {0., -9.81};
 real K = 5.e4;

 real pgrad_x, pgrad_y;

 Thread *pt, *st;/* thread pointers for primary and secondary phases*/

 pt = THREAD_SUB_THREAD(mixture_thread, second_column_phase_index);
 st = THREAD_SUB_THREAD(mixture_thread, first_column_phase_index);

 /* at this point the phase threads are known for primary (0) and
 secondary(1) phases */

 pgrad_x = C_DP(c,mixture_thread)[0];
```

```
pgrad_y = C_DP(c,mixture_thread)[1];

vector_result[0] =
-(pgrad_x/K)
+( ((C_R(c, st)-
C_R(c, pt))/K)*
grav[0]);

vector_result[1] =
-(pgrad_y/K)
+( ((C_R(c, st)-
C_R(c, pt))/K)*
grav[1]);
}
```

i Note that the pressure gradient macro C␣DP is now obsolete. A more current
pressure gradient macro can be found in Table 3.2.4.

Hooking a Vector Exchange Property UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE␣VECTOR␣EXCHANGE␣PROPERTY is in-
terpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs),
the name of the argument that you supplied as the first DEFINE macro argument (e.g.,
custom␣slip) will become visible and selectable in the Phase Interaction dialog box in
ANSYS FLUENT. See Section 6.3.5: Hooking DEFINE␣VECTOR␣EXCHANGE␣PROPERTY UDFs
for details.

2.5 Discrete Phase Model (DPM) DEFINE **Macros**

This section contains descriptions of DEFINE macros for the discrete phase model (DPM). Table 2.5.1 provides a quick reference guide to the DPM DEFINE macros, the functions they define, and the dialog boxes where they are activated in ANSYS FLUENT. Definitions of each DEFINE macro are contained in the udf.h header file. For your convenience, they are listed in Appendix B.

- Section 2.5.1: DEFINE_DPM_BC

- Section 2.5.2: DEFINE_DPM_BODY_FORCE

- Section 2.5.3: DEFINE_DPM_DRAG

- Section 2.5.4: DEFINE_DPM_EROSION

- Section 2.5.5: DEFINE_DPM_HEAT_MASS

- Section 2.5.6: DEFINE_DPM_INJECTION_INIT

- Section 2.5.7: DEFINE_DPM_LAW

- Section 2.5.8: DEFINE_DPM_OUTPUT

- Section 2.5.9: DEFINE_DPM_PROPERTY

- Section 2.5.10: DEFINE_DPM_SCALAR_UPDATE

- Section 2.5.11: DEFINE_DPM_SOURCE

- Section 2.5.12: DEFINE_DPM_SPRAY_COLLIDE

- Section 2.5.13: DEFINE_DPM_SWITCH

- Section 2.5.14: DEFINE_DPM_TIMESTEP

- Section 2.5.15: DEFINE_DPM_VP_EQUILIB

Table 2.5.1: Quick Reference Guide for DPM-Specific DEFINE Macros

Function	DEFINE Macro	Dialog Box Activated In
particle state at boundaries	DEFINE_DPM_BC	boundary condition (e.g., Velocity Inlet)
body forces on particles	DEFINE_DPM_BODY_FORCE	Discrete Phase Model
drag coefficients between particles and fluid	DEFINE_DPM_DRAG	Discrete Phase Model
erosion and accretion rates	DEFINE_DPM_EROSION	Discrete Phase Model
heat and mass transfer of multicomponent particles to the gas phase	DEFINE_DPM_HEAT_MASS	Set Injection Properties
initializes injections	DEFINE_DPM_INJECTION_INIT	Set Injection Properties
custom laws for particles	DEFINE_DPM_LAW	Custom Laws
modifies what is written to the sampling plane output	DEFINE_DPM_OUTPUT	Sample Trajectories
material properties	DEFINE_DPM_PROPERTY	Create/Edit Materials
updates scalar every time a particle position is updated	DEFINE_DPM_SCALAR_UPDATE	Discrete Phase Model
particle source terms	DEFINE_DPM_SOURCE	Discrete Phase Model
particle collisions algorithm	DEFINE_DPM_SPRAY_COLLIDE	Discrete Phase Model
changes the criteria for switching between laws	DEFINE_DPM_SWITCH	Custom Laws
time step control for DPM simulation	DEFINE_DPM_TIMESTEP	Discrete Phase Model
equilibrium vapor pressure of vaporizing components of multicomponent particles	DEFINE_DPM_VP_EQUILIB	Create/Edit Materials

2.5.1 DEFINE_DPM_BC

Description

You can use DEFINE_DPM_BC to specify your own boundary conditions for particles. The function is executed every time a particle touches a boundary of the domain, except for symmetric or periodic boundaries. You can define a separate UDF (using DEFINE_DPM_BC) for each boundary.

Usage

DEFINE_DPM_BC(name,p,t,f,f_normal,dim)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
Thread *t	Pointer to the face thread the particle is currently hitting.
face_t f	Index of the face that the particle is hitting.
real f_normal[]	Array that contains the unit vector which is normal to the face.
int dim	Dimension of the flow problem. The value is 2 in 2d, for 2d-axisymmetric and 2d-axisymmetric-swirling flow, while it is 3 in 3d flows.

Function returns
int

There are six arguments to DEFINE_DPM_BC: name, p, t, f, f_normal, and dim. You supply name, the name of the UDF. p, t, f, f_normal, and dim are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the new velocity of a particle after hitting the wall, and then return the status of the particle track (as an int), after it has hit the wall.

i Pointer p can be used as an argument to the particle-specific macros (defined in Section 3.2.7: DPM Macros) to obtain information about particle properties.

Example 1

This example shows the usage of DEFINE_DPM_BC for a simple reflection at walls. It is similar to the reflection method executed by ANSYS FLUENT except that ANSYS

FLUENT accommodates moving walls. The function must be executed as a compiled UDF.

The function assumes an ideal reflection for the normal velocity component (nor_coeff = 1) while the tangential component is damped (tan_coeff = 0.3). First, the angle of incidence is computed. Next, the normal particle velocity, with respect to the wall, is computed and subtracted from the particles velocity. The reflection is complete after the reflected normal velocity is added. The new particle velocity has to be stored in P_VEL0 to account for the change of particle velocity in the momentum balance for coupled flows. The function returns PATH_ACTIVE for inert particles while it stops particles of all other types.

```
/* reflect boundary condition for inert particles */
#include "udf.h"
DEFINE_DPM_BC(bc_reflect,p,t,f,f_normal,dim)
{
  real alpha;  /* angle of particle path with face normal */
  real vn=0.;
  real nor_coeff = 1.;
  real tan_coeff = 0.3;
  real normal[3];
  int i, idim = dim;
  real NV_VEC(x);

#if RP_2D
  /* dim is always 2 in 2D compilation. Need special treatment for 2d
     axisymmetric and swirl flows */
  if (rp_axi_swirl)
    {
      real R = sqrt(P_POS(p)[1]*P_POS(p)[1] +
                    P_POS(p)[2]*P_POS(p)[2]);
      if (R > 1.e-20)
        {
          idim = 3;
          normal[0] = f_normal[0];
          normal[1] = (f_normal[1]*P_POS(p)[1])/R;
          normal[2] = (f_normal[1]*P_POS(p)[2])/R;
        }
      else
        {
          for (i=0; i<idim; i++)
            normal[i] = f_normal[i];
        }
    }
```

```
      else
#endif
    for (i=0; i<idim; i++)
      normal[i] = f_normal[i];

  if(p->type==DPM_TYPE_INERT)
    {
      alpha = M_PI/2. - acos(MAX(-1.,MIN(1.,NV_DOT(normal,P_VEL(p))/
                                  MAX(NV_MAG(P_VEL(p)),DPM_SMALL))));
      if ((NNULLP(t)) && (THREAD_TYPE(t) == THREAD_F_WALL))
        F_CENTROID(x,f,t);

      /* calculate the normal component, rescale its magnitude by
         the coefficient of restitution and subtract the change */

      /* Compute normal velocity. */
      for(i=0; i<idim; i++)
        vn += P_VEL(p)[i]*normal[i];

      /* Subtract off normal velocity. */
      for(i=0; i<idim; i++)
        P_VEL(p)[i] -= vn*normal[i];

      /* Apply tangential coefficient of restitution. */
      for(i=0; i<idim; i++)
        P_VEL(p)[i] *= tan_coeff;

      /* Add reflected normal velocity. */
      for(i=0; i<idim; i++)
        P_VEL(p)[i] -= nor_coeff*vn*normal[i];

      /* Store new velocity in P_VEL0 of particle */
      for(i=0; i<idim; i++)
        P_VEL0(p)[i] = P_VEL(p)[i];

      return PATH_ACTIVE;
    }

  return PATH_ABORT;
}
```

Example 2

This example shows how to use DEFINE_DPM_BC for a wall impingement model. The function must be executed as a compiled UDF.

```
#include "udf.h"
#include "dpm.h"
#include "surf.h"
#include "random.h"

/* define a user-defined dpm boundary condition routine
 * bc_reflect: name
 * p:          the tracked particle
 * t:          the touched face thread
 * f:          the touched face
 * f_normal:   normal vector of touched face
 * dim:        dimension of the problem (2 in 2d and 2d-axi-swirl, 3 in 3d)
 *
 * return is the status of the particle, see enumeration of Path_Status
 * in dpm.h
 */

#define V_CROSS(a,b,r)\
        ((r)[0] = (a)[1]*(b)[2] - (b)[1]*(a)[2],\
         (r)[1] = (a)[2]*(b)[0] - (b)[2]*(a)[0],\
         (r)[2] = (a)[0]*(b)[1] - (b)[0]*(a)[1])


DEFINE_DPM_BC(bc_wall_jet, p, thread, f, f_normal, dim)
{
  /*
     Routine implementing the Naber and Reitz Wall
     impingement model (SAE 880107)
  */

  real normal[3];
  real tan_1[3];
  real tan_2[3];
  real rel_vel[3];
  real face_vel[3];

  real alpha, beta, phi, cp, sp;
  real rel_dot_n, vmag, vnew, dum;
```

```
    real weber_in, weber_out;

    int i, idim = dim;

    cxboolean moving = (SV_ALLOCATED_P (thread,SV_WALL_GRID_V) &&
      SV_ALLOCATED_P (thread,SV_WALL_V      )    );

  #if RP_2D
    if (rp_axi_swirl)
      {
        real R = sqrt(P_POS(p)[1]*P_POS(p)[1] +
      P_POS(p)[2]*P_POS(p)[2]);

        if (R > 1.e-20)
{
  idim = 3;
  normal[0] = f_normal[0];
  normal[1] = (f_normal[1]*P_POS(p)[1])/R;
  normal[2] = (f_normal[1]*P_POS(p)[2])/R;
}
        else
{
  for (i=0; i<idim; i++)
    normal[i] = f_normal[i];
}
      }
    else
  #endif
      for (i=0; i<idim; i++)
        normal[i] = f_normal[i];

    /*
       Set up velocity vectors and calculate the Weber number
       to determine the regime.
    */

    for (i=0; i < idim; i++)
      {
        if (moving)
face_vel[i] = WALL_F_VV(f,thread)[i] + WALL_F_GRID_VV(f,thread)[i];
        else
face_vel[i] = 0.0;
```

```
      rel_vel[i] = P_VEL(p)[i] - face_vel[i];
    }

  vmag = MAX(NV_MAG(rel_vel),DPM_SMALL);

  rel_dot_n = MAX(NV_DOT(rel_vel,normal),DPM_SMALL);

  weber_in = P_RHO(p) * SQR(rel_dot_n) * P_DIAM(p) /
     MAX(DPM_SURFTEN(p), DPM_SMALL);

  /*
     Regime where bouncing occurs (We_in < 80).
     (Data from Mundo, Sommerfeld and Tropea
      Int. J. of Multiphase Flow, v21, #2, pp151-173, 1995)
  */

  if (weber_in <= 80.)
    {
      weber_out = 0.6785*weber_in*exp(-0.04415*weber_in);
      vnew = rel_dot_n * (1.0 + sqrt( weber_out /
         MAX( weber_in, DPM_SMALL )));

      /*
 The normal component of the velocity is changed based
 on the experimental paper above (i.e. the Weber number
 is based on the relative velocity).
      */

      for (i=0; i < idim; i++)
P_VEL(p)[i] = rel_vel[i] - vnew*normal[i] + face_vel[i];

    }

  if (weber_in > 80.)
    {
      alpha = acos(-rel_dot_n/vmag);

      /*
         Get one tangent vector by subtracting off the normal
         component from the impingement vector, then cross the
         normal with the tangent to get an out of plane vector.
      */
```

```
      for (i=0; i < idim; i++)
  tan_1[i] = rel_vel[i] - rel_dot_n*normal[i];

      UNIT_VECT(tan_1,tan_1);

      V_CROSS(tan_1,normal,tan_2);

      /*
         beta is calculated by neglecting the coth(alpha)
         term in the paper (it is approximately right).
      */

      beta = MAX(M_PI*sqrt(sin(alpha)/(1.0-sin(alpha))),DPM_SMALL);

      phi= -M_PI/beta*log(1.0-cheap_uniform_random()*(1.0-exp(-beta)));

      if (cheap_uniform_random() > 0.5)
  phi = -phi;

      vnew = vmag;

      cp = cos(phi);
      sp = sin(phi);

      for (i=0; i < idim; i++)
  P_VEL(p)[i] = vnew*(tan_1[i]*cp + tan_2[i]*sp) + face_vel[i];

    }

  /*
    Subtract off from the original state.
  */
  for (i=0; i < idim; i++)
    P_VEL0(p)[i] = P_VEL(p)[i];

  if ( DPM_STOCHASTIC_P(P_INJECTION(p)) )
    {

      /* Reflect turbulent fluctuations also */
      /* Compute normal velocity. */

      dum = 0;
      for(i=0; i<idim; i++)
```

```
dum += p->V_prime[i]*normal[i];

      /* Subtract off normal velocity. */

      for(i=0; i<idim; i++)
p->V_prime[i] -= 2.*dum*normal[i];
    }
  return PATH_ACTIVE;
}
```

Hooking a DPM Boundary Condition UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_BC is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the appropriate boundary condition dialog box (e.g., the Velocity Inlet dialog box) in ANSYS FLUENT. See Section 6.4.1: Hooking DEFINE_DPM_BC UDFs for details on how to hook your DEFINE_DPM_BC UDF to ANSYS FLUENT.

2.5.2 DEFINE_DPM_BODY_FORCE

Description

You can use DEFINE_DPM_BODY_FORCE to specify a body force other than a gravitational or drag force on the particles.

Usage

DEFINE_DPM_BODY_FORCE(name,p,i)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
int i	An index (0, 1, or 2) that identifies the Cartesian component of the body force that is to be returned by the function.

Function returns
real

There are three arguments to DEFINE_DPM_BODY_FORCE: name, p, and i. You supply name, the name of the UDF. p and i are variables that are passed by the ANSYS FLUENT solver

to your UDF. Your UDF will need to return the `real` value of the acceleration due to the body force (in m/s^2) to the ANSYS FLUENT solver.

> *i* Pointer `p` can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

The following UDF, named `particle_body_force`, computes the magnetic force on a charged particle. DEFINE_DPM_BODY_FORCE is called at every particle time step in ANSYS FLUENT and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

In the UDF presented below a charged particle is introduced upstream, into a laminar flow, and travels downstream until t=tstart when a magnetic field is applied. The particle takes on an approximately circular path (not an exact circular path, because the speed and magnetic force vary as the particle is slowed by the surrounding fluid).

The macro `P_TIME(p)` gives the current time for a particle traveling along a trajectory, which is pointed to by `p`.

```
/* UDF for computing the magnetic force on a charged particle */

#include "udf.h"

#define Q 1.0          /* particle electric charge      */
#define BZ 3.0         /* z component of magnetic field */
#define TSTART 18.0    /* field applied at t = tstart   */

/* Calculate magnetic force on charged particle.  Magnetic   */
/* force is particle charge times cross product of particle  */
/* velocity with magnetic field: Fx= q*bz*Vy,  Fy= -q*bz*Vx  */

DEFINE_DPM_BODY_FORCE(particle_body_force,p,i)
{
        real bforce=0;
        if(P_TIME(p)>=TSTART)
          {
           if(i==0) bforce=Q*BZ*P_VEL(p)[1];

           else if(i==1) bforce=-Q*BZ*P_VEL(p)[0];

          }
```

```
        else
            bforce=0.0;
        /* an acceleration should be returned */
        return (bforce/P_MASS(p));
}
```

Hooking a DPM Body Force UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_BODY_FORCE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Discrete Phase Model dialog box in ANSYS FLUENT. See Section 6.4.2: Hooking DEFINE_DPM_BODY_FORCE UDFs for details on how to hook your DEFINE_DPM_BODY_FORCE UDF to ANSYS FLUENT.

2.5.3 DEFINE_DPM_DRAG

Description

You can use DEFINE_DPM_DRAG to specify the drag coefficient, C_D, between particles and fluid defined by the following equation:

$$F_D = \frac{18\mu}{\rho_p D_p^2} \frac{C_D \mathrm{Re}}{24}$$

Usage

DEFINE_DPM_DRAG(name,Re,p)

Argument Type	Description
symbol name	UDF name.
real Re	particle Reynolds number based on the particle diameter and relative gas velocity.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.

Function returns
real

There are three arguments to DEFINE_DPM_DRAG: name, Re, and p. You supply name, the name of the UDF. Re and p are variables that are passed by the ANSYS FLUENT solver

to your UDF. Your UDF will need to return the `real` value of the drag force on a particle. The value returned to the solver must be dimensionless and represent `18 * Cd * Re / 24`.

> *i*   Pointer `p` can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

The following UDF, named `particle_drag_force`, computes the drag force on a particle and is a variation of the body force UDF presented in Section 2.5.2: DEFINE DPM BODY FORCE. The flow is the same, but a different curve is used to describe the particle drag. DEFINE DPM DRAG is called at every particle time step in ANSYS FLUENT, and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

```
/**************************************************************************
   UDF for computing particle drag coefficient (18 Cd Re/24)
   curve as suggested by R. Clift, J. R. Grace and M.E. Weber
   "Bubbles, Drops, and Particles" (1978)
**************************************************************************/

#include "udf.h"

DEFINE_DPM_DRAG(particle_drag_force,Re,p)
{
  real w, drag_force;

  if (Re < 0.01)
    {
    drag_force=18.0;
    return (drag_force);
    }
  else if (Re < 20.0)
    {
    w = log10(Re);
    drag_force = 18.0 + 2.367*pow(Re,0.82-0.05*w) ;
    return (drag_force);
    }
  else
    /* Note: suggested valid range 20 < Re < 260 */
    {
```

```
    drag_force = 18.0 + 3.483*pow(Re,0.6305) ;
    return (drag_force);
    }
}
```

Hooking a DPM Drag Coefficient UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_DRAG is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Discrete Phase Model dialog box in ANSYS FLUENT. See Section 6.4.3: Hooking DEFINE_DPM_DRAG UDFs for details on how to hook your DEFINE_DPM_DRAG UDF to ANSYS FLUENT.

2.5.4 DEFINE_DPM_EROSION

Description

You can use DEFINE_DPM_EROSION to specify the erosion and accretion rates calculated as the particle stream strikes a wall surface. The function is called when the particle encounters a reflecting surface.

Usage

DEFINE_DPM_EROSION(name,p,t,f,normal,alpha,Vmag,mdot)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
Thread *t	Pointer to the face thread the particle is currently hitting.
face_t f	Index of the face that the particle is hitting.
real normal[]	Array that contains the unit vector that is normal to the face.
real alpha	Variable that represents the impact angle between the particle path and the face (in radians).
real Vmag	Variable that represents the magnitude of the particle velocity (in m/s).
real mdot	Flow rate of the particle stream as it hits the face (in kg/s).

Function returns
void

There are eight arguments to `DEFINE_DPM_EROSION`: `name`, `p`, `t`, `f`, `normal`, `alpha`, `Vmag`, and `mdot`. You supply `name`, the name of the UDF. `p`, `t`, `f`, `normal`, `alpha`, `Vmag`, and `mdot` are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the values for the erosion rate and/or accretion rate and store the values at the faces in `F_STORAGE_R(f,t,SV_DPMS_EROSION)` and `F_STORAGE_R(f,t,SV_DPMS_ACCRETION)`, respectively.

> *i* Pointer `p` can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

The following is an example of a compiled UDF that uses `DEFINE_DPM_EROSION` to extend postprocessing of wall impacts in a 2D axisymmetric flow. It provides additional information on how the local particle deposition rate depends on the diameter and normal velocity of the particles. It is based on the assumption that every wall impact leads to more accretion, and, therefore, every trajectory is "evaporated" at its first wall impact. (This is done by first setting a DPM user scalar within `DEFINE_DPM_EROSION`, which is then evaluated within `DEFINE_DPM_LAW`, where `P_MASS` is set to zero.) User-defined memory locations (UDMLs) are used to store and visualize the following:

- number of wall impacts since UDMLs were reset. (Resetting is typically done at the beginning of an ANSYS FLUENT session by the use of `DEFINE_ON_DEMAND` in order to avoid the use of uninitialized data fields. Resetting prevents the addition of sampled data being read from a file).

- average diameter of particles hitting the wall.

- average radial velocity of particles.

Before tracing the particles, you will have to reset the UDMLs and assign the global domain pointer by executing the `DEFINE_ON_DEMAND` function.

```
/******************************************************************************
   UDF for extending postprocessing of wall impacts
******************************************************************************/
#include "udf.h"

#define MIN_IMPACT_VELO -1000.
  /* Minimum particle velocity normal to wall (m/s) to allow Accretion.*/

Domain *domain; /* Get the domain pointer and assign it later to domain*/
```

```
enum            /* Enumeration of used User-Defined Memory Locations. */
{
  NUM_OF_HITS,     /* Number of particle hits into wall face considered.*/
  AVG_DIAMETER,    /* Average diameter of particles that hit the wall. */
  AVG_RADI_VELO,    /* Average radial velocity of ""  ""  ------------ */
  NUM_OF_USED_UDM
};

int UDM_checked = 0;     /* Availability of UDMLs checked? */

void reset_UDM_s(void);  /* Function to follow below. */

int check_for_UDM(void)     /* Check for UDMLs' availability... */
{
  Thread *t;

  if (UDM_checked)
    return UDM_checked;

/* if (!rp_axi)*/
 /* Internal_Error("UDF-Error: only valid for 2d-axisymmetric cases!\n");*/
  thread_loop_c(t,domain)    /* We require all cell threads to */
    {                          /* provide space in memory for UDML */
      if (FLUID_THREAD_P(t))
        if (NULLP(THREAD_STORAGE(t,SV_UDM_I)))
            return 0;
    }

  UDM_checked = 1;    /* To make the following work properly... */
  reset_UDM_s();       /* This line will be executed only once, */
  return UDM_checked;  /* because check_for_UDM checks for */
}                       /* UDM_checked first. */

void
reset_UDM_s(void)
{
  Thread *t;
  cell_t  c;
  face_t  f;
  int     i;

  if (!check_for_UDM())  /* Don't do it, if memory is not available. */
```

```
     return;

  Message("Resetting User Defined Memory...\n");

  thread_loop_f(t, domain)
    {
      if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
        {
          begin_f_loop(f,t)
            {
              for (i = 0; i < NUM_OF_USED_UDM; i++)
                F_UDMI(f,t,i) = 0.;
            }
          end_f_loop(f, t)
        }
      else
        {
          Message("Skipping FACE thread no. %d..\n", THREAD_ID(t));
        }
    }
  thread_loop_c(t,domain)
    {
      if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
        {
          begin_c_loop(c,t)
            {
              for (i = 0; i < NUM_OF_USED_UDM; i++)
                C_UDMI(c,t,i) = 0.;
            }
          end_c_loop(c,t)
        }
      else
        {
          Message(" Skipping CELL thread no. %d..\n", THREAD_ID(t));
        }
    }                  /* Skipping Cell Threads can happen if the user */
                          /* uses reset_UDM prior to initializing. */
  Message(" --- Done.\n");
}

DEFINE_DPM_SCALAR_UPDATE(dpm_scalup,c,t,if_init,p)
{  if (if_init)
     P_USER_REAL(p, 0) = 0;    /* Simple initialization. Used later for
```

```
                                        stopping trajectory calculation */
}

DEFINE_DPM_EROSION(dpm_accr, p, t, f, normal, alpha, Vmag, Mdot)
{
  real A[ND_ND], area;
  int num_in_data;
  Thread *t0;
  cell_t  c0;

  real radi_pos[2], radius, imp_vel[2], vel_ortho;

/* The following is ONLY valid for 2d-axisymmetric calculations!!! */
/* Additional effort is necessary because DPM tracking is done in  */
/* THREE dimensions for TWO-dimensional axisymmetric calculations. */

  radi_pos[0] = P_POS(p)[1];      /* Radial location vector. */
  radi_pos[1] = P_POS(p)[2];      /* (Y and Z in 0 and 1...) */

  radius = NV_MAG(radi_pos);
  NV_VS(radi_pos, =, radi_pos, /, radius);
                                  /* Normalized radius direction vector.*/
  imp_vel[0] = P_VEL(p)[0];    /* Axial particle velocity component. */
  imp_vel[1] = NVD_DOT(radi_pos, P_VEL(p)[1], P_VEL(p)[2], 0.);
  /* Dot product of normalized radius vector and y & z components */
  /* of particle velocity vector gives _radial_ particle velocity */
  /* component  */
  vel_ortho = NV_DOT(imp_vel, normal); /*velocity orthogonal to wall */

  if (vel_ortho < MIN_IMPACT_VELO)  /* See above, MIN_IMPACT_VELO */
    return;

  if (!UDM_checked)        /* We will need some UDMs, */
    if (!check_for_UDM()) /* so check for their availability.. */
      return;             /* (Using int variable for speed, could */
                          /*  even just call check_for UDFM().) */
  c0 = F_C0(f,t);
  t0 = THREAD_T0(t);

  num_in_data = F_UDMI(f,t,NUM_OF_HITS);

/* Average diameter of particles that hit the particular wall face:*/
  F_UDMI(f,t,AVG_DIAMETER) = (P_DIAM(p)
```

```
                            +  num_in_data * F_UDMI(f,t,AVG_DIAMETER))
                               / (num_in_data + 1);
    C_UDMI(c0,t0,AVG_DIAMETER) = F_UDMI(f,t,AVG_DIAMETER);

  /* Average velocity normal to wall of particles hitting the wall:*/
    F_UDMI(f,t,AVG_RADI_VELO) = (vel_ortho
                            +  num_in_data * F_UDMI(f,t,AVG_RADI_VELO))
                               / (num_in_data + 1);
    C_UDMI(c0,t0,AVG_RADI_VELO) = F_UDMI(f,t,AVG_RADI_VELO);

    F_UDMI(f, t, NUM_OF_HITS) = num_in_data + 1;
    C_UDMI(c0,t0,NUM_OF_HITS) = num_in_data + 1;

    F_AREA(A,f,t);
    area = NV_MAG(A);
    F_STORAGE_R(f,t,SV_DPMS_ACCRETION) += Mdot / area;
                                            /* copied from source. */

    P_USER_REAL(p,0) = 1.;      /* "Evaporate" */
}

DEFINE_DPM_LAW(stop_dpm_law,p,if_cpld)
{
  if (0. < P_USER_REAL(p,0))
    P_MASS(p) = 0.;            /* "Evaporate" */
}

DEFINE_ON_DEMAND(reset_UDM)
{
  /* assign domain pointer with global domain */
  domain = Get_Domain(1);
  reset_UDM_s();
}
```

Hooking an Erosion/Accretion UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_EROSION is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the **Discrete Phase Model** dialog box in ANSYS FLUENT. See Section 6.4.4: Hooking DEFINE_DPM_EROSION UDFs for details on how to hook your DEFINE_DPM_EROSION UDF to ANSYS FLUENT.

2.5.5 DEFINE_DPM_HEAT_MASS

Description

You can use DEFINE_DPM_HEAT_MASS to specify the heat and mass transfer of multicomponent particles to the gas phase.

Usage

DEFINE_DPM_HEAT_MASS(name,p,C_p,hgas,hvap,cvap_surf, Z, dydt,dzdt)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
real C_p	Particle heat capacity.
real *hgas	Enthalpies of vaporizing gas phase species.
real *hvap	Vaporization enthalpies of vaporizing components.
real *cvap_surf	Vapor equilibrium concentrations of vaporizing components.
real Z	Compressibility, Z^V
real *dydt	Source terms of the particle temperature and component masses.
dpms_t *dzdt	Source terms of the gas phase enthalpy and species masses.

Function returns
void

There are eight arguments to DEFINE_DPM_HEAT_MASS: name,e,p,C_p,hgas,hvap,cvap_surf, dydt,and dzdt. You supply name, the name of the UDF. e,p,C_p,hgas,hvap,and cvap_surf are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the particle and gas phase source terms and store the values in dydt and dzdt, respectively.

Example

The following is an example of a compiled UDF that uses DEFINE_DPM_HEAT_MASS. It implements the source terms for the following:

Source Term	Variable	Unit
particle temperature	dydt[0]	K/s
particle component mass	dydt[1..]	kg/s
gas phase enthalpy	dzdt->energy	J/s
gas phase species mass	dzdt->species[0..]	kg/s

```
/*****************************************************************************
   UDF for defining the heat and mass transport for
   multicomponent particle vaporization
*****************************************************************************/
#include "udf.h"

DEFINE_DPM_HEAT_MASS(multivap,p,Cp,hgas,hvap,cvap_surf,Z,dydt,dzdt)
{
 int ns;
 Material *sp;

 real dens_total = 0.0;                     /* total vapor density*/
 real P_total = 0.0;                        /* vapor pressure */
 int nc = TP_N_COMPONENTS(p);               /* number of particle components */
 Thread *t0 = P_CELL_THREAD(p);             /* thread where the particle is in*/
 Material *gas_mix = THREAD_MATERIAL(DPM_THREAD(t0, p)); /* gas mixture
  material */
 Material *cond_mix = P_MATERIAL(p);  /* particle mixture material*/
 cphase_state_t *c = &(p->cphase); /* cell information of particle location*/
 real molwt[MAX_SPE_EQNS];    /* molecular weight of gas species */
 real Tp = P_T(p);            /* particle temperature */
 real mp = P_MASS(p);         /* particle mass */
 real molwt_bulk = 0.;        /* average molecular weight in bulk gas */
 real Dp = DPM_DIAM_FROM_VOL(mp / P_RHO(p));    /* particle diameter */
 real Ap = DPM_AREA(Dp);                        /* particle surface */
 real Pr = c->sHeat * c->mu / c->tCond;         /* Prandtl number */
 real Nu = 2.0 + 0.6 * sqrt(p->Re) * pow(Pr, 1./3.);  /* Nusselt number */
 real h = Nu * c->tCond / Dp;                   /* Heat transfer coefficient*/
 real dh_dt = h * (c->temp - Tp) * Ap;        /* heat source term*/

 dydt[0] += dh_dt / (mp * Cp);
 dzdt->energy -= dh_dt;

 mixture_species_loop(gas_mix,sp,ns)
   {
     molwt[ns] = MATERIAL_PROP(sp,PROP_mwi);    /* molecular weight of gas
          species */
     molwt_bulk += c->yi[ns] / molwt[ns];    /* average molecular weight */
   }

 /* prevent division by zero */
 molwt_bulk = MAX(molwt_bulk,DPM_SMALL);
```

```
for (ns = 0; ns < nc; ns++)
  {
    int gas_index = TP_COMPONENT_INDEX_I(p,ns);   /* gas species index of
     vaporization */
    if( gas_index >= 0 )
      {
        /* condensed material */
        Material * cond_c = MIXTURE_COMPONENT(cond_mix, ns);
        /* vaporization temperature */
        real vap_temp = MATERIAL_PROP(cond_c,PROP_vap_temp);
        /* diffusion coefficient */
        real D = MATERIAL_PROP_POLYNOMIAL(cond_c, PROP_binary_diffusivity,
 c->temp);
        /* Schmidt number */
        real Sc = c->mu / ( c->rho * D );
        /* mass transfer coefficient */
        real k = (2. + 0.6 * sqrt(p->Re) * pow(Sc, 1./3.)) * D / Dp;
        /* bulk gas concentration (ideal gas) */
        real cvap_bulk = c->pressure / UNIVERSAL_GAS_CONSTANT / c->temp
          * c->yi[gas_index] / molwt_bulk / solver_par.molWeight[gas_index];
        /* vaporization rate */
        real vap_rate = k * molwt[gas_index] * Ap
          * (cvap_surf[ns] - cvap_bulk);

        /* no vaporization below vaporization temperature, no condensation */
        if (Tp < vap_temp || vap_rate < 0.0)
          vap_rate = 0.;

        dydt[1+ns] -= vap_rate;
        dzdt->species[gas_index] += vap_rate;
        /* dT/dt = dh/dt / (m Cp)*/
        dydt[0] -= hvap[gas_index] * vap_rate / ( mp * Cp );
        /* gas enthalpy source term */
        dzdt->energy += hgas[gas_index] * vap_rate;

        P_total += cvap_surf[ns];
        dens_total += cvap_surf[ns] * molwt[gas_index];
      }
  }

/* multicomponent boiling */
P_total *= Z * UNIVERSAL_GAS_CONSTANT * Tp;
```

```
if (P_total > c->pressure && dydt[0] > 0.)
  {
    real h_boil = dydt[0] * mp * Cp;
    /* keep particle temperature constant */
    dydt[0] = 0.;
    for (ns = 0; ns < nc; ns++)
      {
        int gas_index = TP_COMPONENT_INDEX_I(p,ns);
        if (gas_index >= 0)
          {
            real boil_rate = h_boil / hvap[gas_index] * cvap_surf[ns] *
    molwt[gas_index] / dens_total;
            /* particle component mass source term */
            dydt[1+ns] -= boil_rate;
            /* fluid species source */
            dzdt->species[gas_index] += boil_rate;
            /* fluid energy source */
            dzdt->energy += hgas[gas_index] * boil_rate;
          }
      }
  }
}
```

Hooking a DPM Particle Heat and Mass Transfer UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_HEAT_MASS is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., multivap) will become visible in the Set Injection Properties dialog box in ANSYS FLUENT. See Section 6.4.5: Hooking DEFINE_DPM_HEAT_MASS UDFs for details on how to hook your DEFINE_DPM_HEAT_MASS UDF to ANSYS FLUENT.

2.5.6 DEFINE_DPM_INJECTION_INIT

Description

You can use DEFINE_DPM_INJECTION_INIT to initialize a particle's injection properties such as location, diameter, and velocity.

Usage

DEFINE_DPM_INJECTION_INIT(name,I)

Argument Type	Description
symbol name	UDF name.
Injection *I	Pointer to the Injection structure which is a container for the particles being created. This function is called twice for each Injection before the first DPM iteration, and then called once for each Injection before the particles are injected into the domain at each subsequent DPM iteration.

Function returns
void

There are two arguments to DEFINE_DPM_INJECTION_INIT: name and I. You supply name, the name of the UDF. I is a variable that is passed by the ANSYS FLUENT solver to your UDF.

Example

The following UDF, named init_bubbles, initializes particles on a surface injection due to a surface reaction. This function must be executed as a compiled UDF and can be used *only* on UNIX and Linux systems. Note that if you are going to use this UDF in a transient simulation to compute transient particles, you will need to replace loop(p, I->p) with loop(p, I->p_init). Transient particle initialization cannot be performed with a loop over I->p.

```
/**********************************************************************
   UDF that initializes particles on a surface injection due
   to a surface reaction
**********************************************************************/

#include "udf.h"
#include "surf.h"  /* RP_CELL and RP_THREAD are defined in surf.h */

#define REACTING_SURFACE_ID 2
#define MW_H2 2
#define STOIC_H2 1

/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0

real arrhenius_rate(real temp)
{
 return
PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*temp));
}

/* Species numbers. Must match order in ANSYS FLUENT dialog box */
#define HF  0

/* Reaction Exponents */
#define HF_EXP  2.0



/* Reaction Rate Routine used in UDF */

real reaction_rate(cell_t c, Thread *cthread,real mw[],real yi[])

/* Note that all arguments in the reaction_rate function
call in your .c source file MUST be on the same line or a
compilation error will occur */

{
 real concenHF = C_R(c,cthread)*yi[HF]/mw[HF];

 return arrhenius_rate(C_T(c,cthread))*pow(concenHF,HF_EXP);
}
```

```
real contact_area(cell_t c,Thread *t,int s_id,int *n);



DEFINE_DPM_INJECTION_INIT(init_bubbles,I)
{
 int count,i;
 real area, mw[MAX_SPE_EQNS], yi[MAX_SPE_EQNS];
   /*  MAX_SPE_EQNS is an ANSYS FLUENT constant in materials.h  */

 Particle *p;
 cell_t cell;
 Thread *cthread;
 Material *mix, *sp;

 Message("Initializing Injection: %s\n",I->name);

 loop(p,I->p)  /* Standard ANSYS FLUENT Looping Macro to get particle
                  streams in an Injection */
 {
  cell = P_CELL(p);   /* Get the cell and thread that the particle
                         is currently in   */
  cthread = P_CELL_THREAD(p);

  /* Set up molecular weight & mass fraction arrays */
  mix = THREAD_MATERIAL(cthread);
  mixture_species_loop(mix,sp,i)
  {
   mw[i] = MATERIAL_PROP(sp,PROP_mwi);
   yi[i] = C_YI(cell,cthread,i);
  }

  area = contact_area(cell, cthread, REACTING_SURFACE_ID,&count);
   /* Function that gets total area of REACTING_SURFACE faces in
      contact with cell */
   /* count is the number of contacting faces, and is needed
      to share the total bubble emission between the faces       */
  if (count > 0)  /* if cell is in contact with REACTING_SURFACE */
  {
   P_FLOW_RATE(p) = (area *MW_H2* STOIC_H2 *
                 reaction_rate(cell, cthread, mw, yi))/
                 (real)count;    /* to get correct total flow
                     rate when multiple faces contact the same cell */
```

```
    P_DIAM(p) = 1e-3;
    P_RHO(p) = 1.0;
    P_MASS(p) = P_RHO(p)*M_PI*pow(P_DIAM(p),3.0)/6.0;
   }
   else
    P_FLOW_RATE(p) = 0.0;
 }
}

real contact_area(cell_t c, Thread *t, int s_id, int *n)
{
 int i = 0;
 real area = 0.0, A[ND_ND];

 *n = 0;
 c_face_loop(c,t,i)
 {
  if(THREAD_ID(C_FACE_THREAD(c,t,i)) == s_id)
  {
  (*n)++;
   F_AREA(A,C_FACE(c,t,i), C_FACE_THREAD(c,t,i));
   area += NV_MAG(A);
  }
 }
return area;
}
```

Hooking a DPM Initialization UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_INJECTION_INIT is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Set Injection Properties dialog box in ANSYS FLUENT.

See Section 6.4.6: Hooking DEFINE_DPM_INJECTION_INIT UDFs for details on how to hook your DEFINE_DPM_INJECTION_INIT UDF to ANSYS FLUENT.

2.5.7 DEFINE_DPM_LAW

Description

You can use DEFINE_DPM_LAW to customize laws for particles. For example your UDF can specify custom laws for heat and mass transfer rates for droplets and combusting particles. Additionally, you can specify custom laws for mass, diameter, and temperature properties as the droplet or particle exchanges mass and energy with its surroundings.

Usage

DEFINE_DPM_LAW(name,p,ci)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
int ci	Variable that indicates whether the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

Function returns
void

There are three arguments to DEFINE_DPM_LAW: name, p, and ci. You supply name, the name of the UDF. p and ci are variables that are passed by the ANSYS FLUENT solver to your UDF.

i   Pointer p can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

The following UDF, named `Evapor_Swelling_Law`, models a custom law for the evaporation swelling of particles. The source code can be interpreted or compiled in ANSYS FLUENT. See Section 2.5.13: Example for another example of `DEFINE_DPM_LAW` usage.

```
/**************************************************************************
   UDF that models a custom law for evaporation swelling of particles
**************************************************************************/

#include "udf.h"

DEFINE_DPM_LAW(Evapor_Swelling_Law,p,ci)
{
  real swelling_coeff = 1.1;

  /* first, call standard evaporation routine to calculate
     the mass and heat transfer                                 */
  VaporizationLaw(p);
  /* compute new particle diameter and density */
  P_DIAM(p) = P_INIT_DIAM(p)*(1. + (swelling_coeff - 1.)*
   (P_INIT_MASS(p)-P_MASS(p))/(DPM_VOLATILE_FRACTION(p)*P_INIT_MASS(p)));
  P_RHO(p) = P_MASS(p) / (3.14159*P_DIAM(p)*P_DIAM(p)*P_DIAM(p)/6);
  P_RHO(p) = MAX(0.1, MIN(1e5, P_RHO(p)));
}
```

Hooking a Custom DPM Law to ANSYS FLUENT

After the UDF that you have defined using `DEFINE_DPM_LAW` is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Custom Laws dialog box in ANSYS FLUENT. See Section 6.4.7: Hooking `DEFINE_DPM_LAW` UDFs for details on how to hook your `DEFINE_DPM_LAW` UDF to ANSYS FLUENT.

2.5.8 DEFINE_DPM_OUTPUT

Description

You can use DEFINE_DPM_OUTPUT to modify what is written to the sampling device output. This function allows access to the variables that are written as a particle passes through a sampler (see Chapter 23: Modeling Discrete Phase in the separate User's Guide for details).

Usage

DEFINE_DPM_OUTPUT(name,header,fp,p,t,plane)

Argument Type	Description
symbol name	UDF name.
int header	Variable that is equal to 1 at the first call of the function before particles are tracked and set to 0 for subsequent calls.
FILE *fp	Pointer to the file to or from which you are writing or reading.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
Thread *t	Pointer to the thread that the particle is passing through if the sampler is represented by a mesh surface. If the sampler is not defined as a mesh surface, then the value of t is NULL.
Plane *plane	Pointer to the Plane structure (see dpm.h) if the sampling device is defined as a planar slice (line in 2d). If a mesh surface is used by the sampler, then plane is NULL.

Function returns
void

There are six arguments to DEFINE_DPM_OUTPUT: name, header, fp, p, t, and plane. You supply name, the name of the UDF. header, fp, p, t, and plane are variables that are passed by the ANSYS FLUENT solver to your UDF. The output of your UDF will be written to the file indicated by fp.

> *i*   Pointer p can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

The following UDF named `discrete_phase_sample` samples the size and velocity of discrete phase particles at selected planes downstream of an injection. For 2d axisymmetric simulations, it is assumed that droplets/particles are being sampled at planes (lines) corresponding to constant x. For 3d simulations, the sampling planes correspond to constant z.

To remove particles from the domain after they have been sampled, change the value of `REMOVE_PARTICLES` to `TRUE`. In this case, particles will be deleted following the time step in which they cross the plane. This is useful when you want to sample a spray immediately in front of an injector and you don't wish to track the particles further downstream.

> i This UDF works with unsteady and steady simulations that include droplet break-up or collisions. Note that the discrete phase must be traced in an unsteady manner.

```
#include "udf.h"
/*********************************************************************/
/* UDF that samples discrete phase size and velocity distributions*/
/* within the domain.                                             */
/*********************************************************************/
#define REMOVE_PARTICLES FALSE

DEFINE_DPM_OUTPUT(discrete_phase_sample,header,fp,p,t,plane)
{

#if RP_2D

  real y;

  if(header)
  {
      par_fprintf_head(fp," #Time[s]   R [m]   X-velocity[m/s]");
      par_fprintf_head(fp," W-velocity[m/s]  R-velocity[m/s]  ");
      par_fprintf_head(fp,"Drop Diameter[m]  Number of Drops   ");
      par_fprintf_head(fp,"Temperature [K]  Initial Diam [m]   ");
      par_fprintf_head(fp,"Injection Time [s]  \n");
  }

  if(NULLP(p))
    return;
```

```
    if (rp_axi && (sg_swirl || rp_ke))
      y = MAX(sqrt(SQR(P_POS(p)[1]) + SQR(P_POS(p)[2])),DPM_SMALL);
    else
      y = P_POS(p)[1];

    par_fprintf(fp,"%d %d  %e %f %f  %f  %f  %e  %e  %f  %e  %f \n",
          P_INJ_ID(P_INJECTION(p)),p->part_id, P_TIME(p),y,P_VEL(p)[0],
          P_VEL(p)[1],P_VEL(p)[2],P_DIAM(p),P_N(p),
          P_T(p), P_INIT_DIAM(p),p->time_of_birth);

#else

  real r, x, y;

  if(header)
  {
    par_fprintf_head(fp,"  #Time[s]  R [m]  x-velocity[m/s]  ");
    par_fprintf_head(fp,"y-velocity[m/s]  z-velocity[m/s]   ");
    par_fprintf_head(fp,"Drop Diameter[m]   Number of Drops  ");
    par_fprintf_head(fp,"Temperature [K]   Initial Diam [m]    ");
    par_fprintf_head(fp,"Injection Time [s]  \n");
  }

  if(NULLP(p))
    return;

  x = P_POS(p)[0];
  y = P_POS(p)[1];
  r = sqrt(SQR(x) + SQR(y));

  par_fprintf(fp,"%d %d %e %f  %f %f  %f %e %e %f %e %f \n",
 P_INJ_ID(P_INJECTION(p)), p->part_id, P_TIME(p), r,P_VEL(p)[0],
        P_VEL(p)[1],P_VEL(p)[2],P_DIAM(p),P_N(p),
        P_T(p), P_INIT_DIAM(p), p->time_of_birth);

#endif

#if REMOVE_PARTICLES
  p->stream_index=-1;
#endif
}
```

Hooking a DPM Output UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_OUTPUT is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Sample Trajectories dialog box in ANSYS FLUENT. See Section 6.4.8: Hooking DEFINE_DPM_OUTPUT UDFs for details on how to hook your DEFINE_DPM_OUTPUT UDF to ANSYS FLUENT.

2.5.9 DEFINE_DPM_PROPERTY

Description

You can use DEFINE_DPM_PROPERTY to specify properties of discrete phase materials. For example, you can model the following dispersed phase properties with this type of UDF:

- particle emissivity

- vapor pressure

- vaporization temperature

- particle scattering factor

- boiling point

- particle viscosity

- particle surface tension

Usage

DEFINE_DPM_PROPERTY(name,c,t,p)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell where the particle is located in the given thread.
Thread *t	Pointer to the thread where the particle is located.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.

Function returns
real

There are four arguments to DEFINE_DPM_PROPERTY: name, c, t, and p. DEFINE_DPM_PROPERTY has the same arguments as the DEFINE_PROPERTY function (described in Section 2.3.16: DEFINE_PROPERTY UDFs), with the addition of the pointer to the Tracked_Particle p. You supply name, the name of the UDF. c, t, and p are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the real value of the discrete phase property and return it to the solver.

> **i** Pointer p can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

In the following example, two discrete phase material property UDFs (named coal_emissivity and coal_scattering, respectively) are concatenated into a single C source file. These UDFs must be executed as compiled UDFs in ANSYS FLUENT.

```
/**********************************************************************
   UDF that specifies discrete phase materials
**********************************************************************/

#include "udf.h"

DEFINE_DPM_PROPERTY(coal_emissivity,c,t,p)
{
  real mp0= P_INIT_MASS(p);
  real mp = P_MASS(p);
  real vf, cf;

  /* get the material char and volatile fractions and store them */
  /* in vf and cf                                                 */
  vf=DPM_VOLATILE_FRACTION(p);
  cf=DPM_CHAR_FRACTION(p);

  if (!(((mp/mp0) >= 1) || ((mp/mp0) <= 0)))
    {
      if ((mp/mp0) < (1-(vf)-(cf)))
        {
          /* only ash left */
          /* vf = cf = 0; */
          return .001;
        }
      else if ((mp/mp0) < (1-(vf)))
```

```
              {
                /* only ash and char left */
                /* cf = 1 - (1-(vf)-(cf))/(mp/mp0); */
                /* vf = 0; */
                return 1.0;
              }

          else
            {
                /* volatiles, char, and ash left */
                /* cf =  (cf)/(mp/mp0); */
                /* vf = 1. - (1.-(vf))/(mp/mp0); */
                return 1.0;
            }
        }
    return 1.0;
}


DEFINE_DPM_PROPERTY(coal_scattering,c,t,p)
{
    real mp0= P_INIT_MASS(p);
    real mp = P_MASS(p);
    real cf, vf;

    /* get the original char and volatile fractions and store them */
    /* in vf and cf                                                 */
    vf=DPM_VOLATILE_FRACTION(p);
    cf=DPM_CHAR_FRACTION(p);

    if (!(((mp/mp0) >= 1) || ((mp/mp0) <= 0)))
      {
        if ((mp/mp0) < (1-(vf)-(cf)))
          {
            /* only ash left */
            /* vf = cf = 0; */
            return 1.1;
          }
        else if ((mp/mp0) < (1-(vf)))
          {
            /* only ash and char left */
            /* cf = 1 - (1-(vf)-(cf))/(mp/mp0); */
            /* vf = 0; */
            return 0.9;
```

```
      }

    else
      {
        /* volatiles, char, and ash left */
        /* cf =  (cf)/(mp/mp0); */
        /* vf = 1. - (1.-(vf))/(mp/mp0); */
        return 1.0;
      }
   }
  return 1.0;
}
```

Hooking a DPM Material Property UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_PROPERTY is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Create/Edit Materials dialog box in ANSYS FLUENT. See Section 6.4.9: Hooking DEFINE_DPM_PROPERTY UDFs for details on how to hook your DEFINE_DPM_PROPERTY UDF to ANSYS FLUENT.

2.5.10 DEFINE_DPM_SCALAR_UPDATE

Description

You can use DEFINE_DPM_SCALAR_UPDATE to update scalar quantities every time a particle position is updated. The function allows particle-related variables to be updated or integrated over the life of the particle. Particle values can be stored in an array associated with the Tracked_Particle (accessed with the macro P_USER_REAL(p,i)). Values calculated and stored in the array can be used to color the particle trajectory.

During ANSYS FLUENT execution, the DEFINE_DPM_SCALAR_UPDATE function is called at the start of particle integration (when initialize is equal to 1) and then after each time step for the particle trajectory integration.

Usage

```
DEFINE_DPM_SCALAR_UPDATE(name,c,t,initialize,p)
```

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell that the particle is currently in.
Thread *t	Pointer to the thread the particle is currently in.
int initialize	Variable that has a value of 1 when the function is called at the start of the particle integration, and 0 thereafter.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.

Function returns
void

There are five arguments to DEFINE_DPM_SCALAR_UPDATE: name, c, t, initialize, and p. You supply name, the name of the UDF. c, t, initialize, and p are variables that are passed by the ANSYS FLUENT solver to your UDF.

i Pointer p can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties). Also, the real array user is available for storage. The size of this array should be set in the Discrete Phase Model dialog box in the Number of Scalars field.

Example

The following compiled UDF computes the melting index along a particle trajectory. The DEFINE_DPM_SCALAR_UPDATE function is called at every particle time step in ANSYS FLUENT and requires a significant amount of CPU time to execute.

The melting index is computed from

$$\text{melting index} = \int_0^t \frac{1}{\mu} dt \tag{2.5-1}$$

Also included in this UDF is an initialization function DEFINE_INIT that is used to initialize the scalar variables. DPM_OUTPUT is used to write the melting index at sample planes and surfaces. The macro NULLP, which expands to ((p) == NULL), checks if its argument is a null pointer.

```
/***********************************************************************
UDF for computing the melting index along a particle trajectory
***********************************************************************/
#include "udf.h"

DEFINE_INIT(melt_setup,domain)
{
  /* if memory for the particle variable titles has not been
   * allocated yet, do it now */
  if (NULLP(user_particle_vars)) Init_User_Particle_Vars();
  /* now set the name and label */
  strcpy(user_particle_vars[0].name,"melting-index");
  strcpy(user_particle_vars[0].label,"Melting Index");
  strcpy(user_particle_vars[1].name,"melting-index-0");
  strcpy(user_particle_vars[1].label,"Melting Index 0");
}


/* update the user scalar variables */
DEFINE_DPM_SCALAR_UPDATE(melting_index,cell,thread,initialize,p)
{
  cphase_state_t *c = &(p->cphase);
  if (initialize)
    {
      /* this is the initialization call, set:
 * P_USER_REAL(p,0) contains the melting index, initialize to 0
 * P_USER_REAL(p,1) contains the viscosity at the start of a time step*/
      P_USER_REAL(p,0) = 0.;
      P_USER_REAL(p,1) = c->mu;
    }
  else
    {
      /* use a trapezoidal rule to integrate the melting index */
      P_USER_REAL(p,0) += P_DT(p) * .5 * (1/P_USER_REAL(p,1) + 1/c->mu);
      /* save current fluid viscosity for start of next step */
      P_USER_REAL(p,1) = c->mu;
    }
}


/* write melting index when sorting particles at surfaces */
DEFINE_DPM_OUTPUT(melting_output,header,fp,p,thread,plane)
{
  char name[100];
  if (header)
```

```
   {
     if (NNULLP(thread))
par_fprintf_head(fp,"(%s %d)\n",THREAD_HEAD(thread)->
 dpm_summary.sort_file_name,11);
     else
par_fprintf_head(fp,"(%s %d)\n",plane->sort_file_name,11);
     par_fprintf_head(fp,"(%10s %10s %10s %10s %10s %10s %10s"
       " %10s %10s %10s %10s %s)\n",
       "X","Y","Z","U","V","W","diameter","T","mass-flow",
       "time","melt-index","name");
   }
  else
   {
     sprintf(name,"%s:%d",P_INJECTION(p)->name,p->part_id);

     /* add P_INJ_ID(P_INJECTION(p)) and part_id for sorting in parallel */
     par_fprintf(fp,
 "%d %d ((%10.6g %10.6g %10.6g %10.6g %10.6g %10.6g "
 "%10.6g %10.6g %10.6g %10.6g %10.6g) %s)\n",
 P_INJ_ID(P_INJECTION(p)), p->part_id,
 P_POS(p)[0], P_POS(p)[1], P_POS(p)[2],
 P_VEL(p)[0], P_VEL(p)[1], P_VEL(p)[2],
 P_DIAM(p), P_T(p), P_FLOW_RATE(p), P_TIME(p),
 P_USER_REAL(p,0), name);
   }
}
```

Hooking a DPM Scalar Update UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_SCALAR_UPDATE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Discrete Phase Model dialog box in ANSYS FLUENT.

See Section 6.4.10: Hooking DEFINE_DPM_SCALAR_UPDATE UDFs for details on how to hook your DEFINE_DPM_SCALAR_UPDATE UDF to ANSYS FLUENT.

2.5.11 DEFINE_DPM_SOURCE

Description

You can use DEFINE_DPM_SOURCE to specify particle source terms. The function allows access to the accumulated source terms for a particle in a given cell before they are added to the mass, momentum, and energy exchange terms for coupled DPM calculations.

Usage

DEFINE_DPM_SOURCE(name,c,t,S, strength,p)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell that the particle is currently in.
Thread *t	Pointer to the thread the particle is currently in.
dpms_t *S	Pointer to the source structure dpms_t, which contains the source terms for the cell.
real strength	Particle number flow rate in particles/second (divided by the number of tries if stochastic tracking is used).
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.

Function returns
void

There are six arguments to DEFINE_DPM_SOURCE: name, c, t, S, strength, and p. You supply name, the name of the UDF. c, t, S, strength, and p are variables that are passed by the ANSYS FLUENT solver to your UDF. The modified source terms, after they have been computed by the function, will be stored in S.

> i Pointer p can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

See Section 2.5.13: Example for an example of DEFINE_DPM_SOURCE usage.

Hooking a DPM Source Term UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_SOURCE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Discrete Phase Model dialog box in ANSYS FLUENT. See Section 6.4.11: Hooking DEFINE_DPM_SOURCE UDFs for details on how to hook your DEFINE_DPM_SOURCE UDF to ANSYS FLUENT.

2.5.12 DEFINE_DPM_SPRAY_COLLIDE

Description

You can use DEFINE_DPM_SPRAY_COLLIDE to side-step the default ANSYS FLUENT spray collision algorithm. When droplets collide they may bounce (in which case their velocity changes) or they may coalesce (in which case their velocity is changed, as well as their diameter and number in the DPM parcel). A spray collide UDF is called during droplet tracking after every droplet time step and requires that Droplet Collision is enabled in the Discrete Phase Model dialog box.

Usage

DEFINE_DPM_SPRAY_COLLIDE(name,tp,p)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *tp	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
Particle *p	Pointer to the Particle data structure where particles p are stored in a linked list.

Function returns
void

There are three arguments to DEFINE_DPM_SPRAY_COLLIDE: name, tp, and p. You supply name, the name of the UDF. tp and p are variables that are passed by the ANSYS FLUENT solver to your UDF. When collision is enabled, this linked list is ordered by the cell that the particle is currently in. As particles from this linked list are tracked, they are copied from the particle list into a Tracked_Particle structure.

Example

The following UDF, named `mean_spray_collide`, is a simple (and non-physical) example that demonstrates the usage of `DEFINE_SPRAY_COLLIDE`. The droplet diameters are assumed to relax to their initial diameter over a specified time `t_relax`. The droplet velocity is also assumed to relax to the mean velocity of all droplets in the cell over the same time scale.

```
/**************************************************************
   DPM Spray Collide Example UDF
**************************************************************/
#include "udf.h"
#include "dpm.h"
#include "surf.h"
DEFINE_DPM_SPRAY_COLLIDE(mean_spray_collide,tp,p)
{
  /* non-physical collision UDF that relaxes the particle */
  /* velocity and diameter in a cell to the mean over the */
  /* specified time scale t_relax */

  const real t_relax = 0.001; /* seconds */

  /* get the cell and Thread that the particle is currently in */
  cell_t c  = P_CELL(tp);
  Thread *t = P_CELL_THREAD(tp);

  /* Particle index for looping over all particles in the cell */
  Particle *pi;

  /* loop over all particles in the cell to find their mass */
  /* weighted mean velocity and diameter */
  int i;
  real u_mean[3]={0.}, mass_mean=0.;
  real d_orig = P_DIAM(tp);
  real decay = 1. - exp(-t_relax);
  begin_particle_cell_loop(pi,c,t)
    {
      mass_mean += P_MASS(pi);
      for(i=0;i<3;i++)
        u_mean[i] += P_VEL(pi)[i]*P_MASS(pi);
    }
  end_particle_cell_loop(pi,c,t)

  /* relax particle velocity to the mean and diameter to the */
```

```
      /* initial diameter over the relaxation time scale t_relax */
      if( mass_mean > 0. )
        {
          for(i=0;i<3;i++)
            u_mean[i] /= mass_mean;
          for(i=0;i<3;i++)
            P_VEL(tp)[i] += decay*( u_mean[i] - P_VEL(tp)[i] );
          P_DIAM(tp) += decay*( P_INIT_DIAM(tp) - P_DIAM(tp) );
          /* adjust the number in the droplet parcel to conserve mass */
          P_N(tp) *= CUB( d_orig/P_DIAM(tp) );
        }
    }
```

Hooking a DPM Spray Collide UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_SPRAY_COLLIDE is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Discrete Phase Model dialog box in ANSYS FLUENT.

See Section 6.4.12: Hooking DEFINE_DPM_SPRAY_COLLIDE UDFs for details on how to hook your DEFINE_DPM_SPRAY_COLLIDE UDF to ANSYS FLUENT.

2.5.13 DEFINE_DPM_SWITCH

Description

You can use DEFINE_DPM_SWITCH to modify the criteria for switching between laws. The function can be used to control the switching between the user-defined particle laws and the default particle laws, or between different user-defined or default particle laws.

Usage

```
DEFINE_DPM_SWITCH(name,p,ci)
```

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
int ci	Variable that indicates if the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

Function returns
void

There are three arguments to DEFINE_DPM_SWITCH: name, p, and ci. You supply name, the name of the UDF. p and ci are variables that are passed by the ANSYS FLUENT solver to your UDF.

> *i* Pointer p can be used as an argument to the macros defined in Section 3.2.7: DPM Macros to obtain information about particle properties (e.g., injection properties).

Example

The following is an example of a compiled UDF that uses DEFINE_DPM_SWITCH to switch between DPM laws using a criterion. The UDF switches to DPM_LAW_USER_1 which refers to condenshumidlaw since only one user law has been defined. The switching criterion is the local humidity which is computed in the domain using a DEFINE_ON_DEMAND function, which again calls the function myHumidity for every cell. In the case where the humidity is greater than 1, condensation is computed by applying a simple mass transfer calculation. Otherwise, one of ANSYS FLUENT's standard laws for Vaporization or Inert Heating are applied, depending on the particle mass. The UDF requires one UDML and needs a species called h2o to compute the local humidity.

```
/***********************************************************************
   Concatenated UDFs for the Discrete Phase Model that includes a
   usage of DPM_SWITCH
***********************************************************************/

#include "udf.h"
#include "dpm.h"
#include "surf.h"  /* for  macros: RP_Cell() & RP_Thread()        */
#include "prop.h"  /* for function: Saturation_Pressure() (of water) */

static real dpm_relax=1.0; /*dpm source relaxation */

real H2O_Saturation_Pressure(real T)
{
  real ratio, aTmTp;

  aTmTp = .01 * (T - 338.15);
  ratio = (647.286/T - 1.) *
    (-7.419242 + aTmTp*(.29721 +
              aTmTp*(-.1155286 +
              aTmTp*(8.685635e-3 +
              aTmTp*(1.094098e-3 +
              aTmTp*(-4.39993e-3 +
              aTmTp*(2.520658e-3 -
              aTmTp*5.218684e-4)))))));
  return (22.089e6 * exp(MIN(ratio,35.)));
}


real myHumidity(cell_t c,Thread *t)
{
  int i;
  Material *m=THREAD_MATERIAL(t), *sp;
  real yi_h2o=0,mw_h2o=1.0;
  real r_mix=0.0;

  if(MATERIAL_TYPE(m)==MATERIAL_MIXTURE)
  {
   mixture_species_loop (m,sp,i)
    {
      r_mix += C_YI(c,t,i)/MATERIAL_PROP(sp,PROP_mwi);

      if (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"h2o") ||
```

```
   (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"H2O")))
      {
 yi_h2o = C_YI(c,t,i);
 mw_h2o = MATERIAL_PROP(sp,PROP_mwi);
      }
   }
  }

  return ((ABS_P(C_P(c,t),op_pres) * yi_h2o / (mw_h2o * r_mix)) /
             H2O_Saturation_Pressure(C_T(c,t))) ;
}


#define CONDENS 1.0e-4

DEFINE_DPM_LAW(condenshumidlaw,p,coupled)
{
 real area;
 real mp_dot;
 cell_t c = P_CELL(p);              /* Get Cell and Thread from */
 Thread *t = P_CELL_THREAD(p);  /* Particle Structure using new macros*/

 area = 4.0* M_PI * (P_DIAM(p)*P_DIAM(p));

 /* Note This law only used if Humidity > 1.0 so mp_dot always positive*/
 mp_dot = CONDENS*sqrt(area)*(myHumidity(c,t)-1.0);

 if(mp_dot>0.0)
 {
   P_MASS(p) = P_MASS(p) + mp_dot*P_DT(p);
   P_DIAM(p) = pow(6.0*P_MASS(p)/(P_RHO(p)* M_PI), 1./3.);
   P_T(p)=C_T(c,t); /* Assume condensing particle is in thermal
           equilibrium with fluid in cell */
 }

}

/* define macro that is not yet standard */

#define C_DPMS_ENERGY(c,t)C_STORAGE_R(c,t,SV_DPMS_ENERGY)

DEFINE_DPM_SOURCE(dpm_source,c,t,S,strength,p)
{
  real mp_dot;
```

```
   Material *sp = P_MATERIAL(p);


   /* mp_dot is the (positive) mass source to the continuous phase */
   /* (Difference in mass between entry and exit from cell)        */
   /* multiplied by strength (Number of particles/s in stream)     */

  mp_dot = (P_MASS0(p) - P_MASS(p)) * strength;

   C_DPMS_YI(c,t,0) += mp_dot*dpm_relax;
   C_DPMS_ENERGY(c,t) -= mp_dot*dpm_relax*
MATERIAL_PROP(sp,PROP_Cp)*(C_T(c,t)-298.15);
   C_DPMS_ENERGY(c,t) -= mp_dot*dpm_relax*
MATERIAL_PROP(sp,PROP_latent_heat);
}


#define UDM_RH 0
#define N_REQ_UDM 1
#define CONDENS_LIMIT 1.0e-10

DEFINE_DPM_SWITCH(dpm_switch,p,coupled)
{
 cell_t c = P_CELL(p);
 Thread *t = P_CELL_THREAD(p);

 if(C_UDMI(c,t,UDM_RH) > 1.0)
  P_CURRENT_LAW(p) = DPM_LAW_USER_1;
 else
 {
  if(P_MASS(p) < CONDENS_LIMIT)
   P_CURRENT_LAW(p) = DPM_LAW_INITIAL_INERT_HEATING;
  else
   P_CURRENT_LAW(p) = DPM_LAW_VAPORIZATION;
 }
}

DEFINE_ADJUST(adj_relhum,domain)
{
 cell_t cell;
 Thread *thread;

 /* set dpm source underrelaxation */
```

```
    dpm_relax = Domainvar_Get_Real(ROOT_DOMAIN_ID,"dpm/relax");

  if(sg_udm<N_REQ_UDM)
     Message("\nNot enough user defined memory allocated. %d required.\n",
             N_REQ_UDM);
  else
  {
   real humidity,min,max;

   min=1e10;
   max=0.0;

   thread_loop_c(thread,domain)
   {
    /* Check if thread is a Fluid thread and has UDMs set up on it */
    if (FLUID_THREAD_P(thread)&& NNULLP(THREAD_STORAGE(thread,SV_UDM_I)))
    {
     begin_c_loop(cell,thread)
      humidity=myHumidity(cell,thread);
      min=MIN(min,humidity);
      max=MAX(max,humidity);
      C_UDMI(cell,thread,UDM_RH)=humidity;
     end_c_loop(cell,thread)
    }
   }
   Message("\nRelative Humidity set in udm-%d",UDM_RH);
   Message(" range:(%f,%f)\n",min,max);
  }/* end if for enough UDSs and UDMs */
}


DEFINE_ON_DEMAND(set_relhum)
{
 adj_relhum(Get_Domain(1));
}
```

Hooking a DPM Switching UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE DPM SWITCH is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Custom Laws dialog box in ANSYS FLUENT. See Section 6.4.13: Hooking DEFINE DPM SWITCH UDFs for details on how to hook your DEFINE DPM SWITCH UDF to ANSYS FLUENT.

2.5.14 DEFINE_DPM_TIMESTEP

Description

You can use DEFINE_DPM_TIMESTEP to change the time step for DPM particle tracking based on user-specified inputs. The time step can be prescribed for special applications where a certain time step is needed. It can also be limited to values that are required to validate physical models.

Usage

DEFINE_DPM_TIMESTEP(name,p,ts)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
real ts	Time step.

Function returns
real

There are three arguments to DEFINE_DPM_TIMESTEP: name, p, and ts. You supply the name of your user-defined function. p and ts are variables that are passed by the ANSYS FLUENT solver to your UDF. Your function will return the real value of the DPM particle timestep to the solver.

Example 1

The following compiled UDF named limit_to_e_minus_four sets the time step to a maximum value of $1e^-4$. If the time step computed by ANSYS FLUENT (and passed as an argument) is smaller than $1e^-4$, then ANSYS FLUENT's time step is returned.

```
/* Time step control UDF for DPM  */

#include "udf.h"
#include "dpm.h"

DEFINE_DPM_TIMESTEP(limit_to_e_minus_four,p,dt)
{
   if (dt > 1.e-4)
     {
/*       p->next_time_step = 1.e-4; */
       return 1.e-4;
     }

   return dt;
}
```

Example 2

The following compiled UDF named `limit_to_fifth_of_prt` computes the particle relaxation time based on the formula:

$$\tau_p = frac{\rho_p d_p^2}{18\mu} \frac{24}{C_D Re_p} \tag{2.5-2}$$

where

$$Re_p = \frac{\rho d_p \|u - u_p\|}{\mu} \tag{2.5-3}$$

The particle time step is limited to a fifth of the particle relaxation time. If the particle time step computed by ANSYS FLUENT (and passed as an argument) is smaller than this value, then ANSYS FLUENT's time step is returned.

```
/* Particle time step control UDF for DPM  */

#include "udf.h"
#include "dpm.h"

DEFINE_DPM_TIMESTEP(limit_to_fifth_of_prt,p,dt)
{
   real drag_factor = 0.;
   real p_relax_time;
   cphase_state_t *c = &(p->cphase);
```

```
    /* compute particle relaxation time */
    if (P_DIAM(p) != 0.0)
      drag_factor = DragCoeff(p) * c->mu / ( P_RHO(p) * P_DIAM(p) * P_DIAM(p));
    else
      drag_factor = 1.;

    p_relax_time = 1./drag_factor;

    /* check the condition and return the time step */
    if (dt > p_relax_time/5.)
      {
        return p_relax_time/5.;
      }

    return dt;
}
```

Hooking a DPM Timestep UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_TIMESTEP is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable for **DPM Timestep** in the **Discrete Phase Model** dialog box in ANSYS FLUENT. See Section 6.4.14: Hooking DEFINE_DPM_TIMESTEP UDFs for details on how to hook your DEFINE_DPM_TIMESTEP UDF to ANSYS FLUENT.

2.5.15 DEFINE_DPM_VP_EQUILIB

Description

You can use DEFINE_DPM_VP_EQUILIB to specify the equilibrium vapor pressure of vaporizing components of multicomponent particles.

Usage

DEFINE_DPM_VP_EQUILIB(name,p,cvap_surf, Z)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *p	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
real *cvap_surf	Array that contains the equilibrium vapor concentration
*Z	Compressibilty, Z
	over the particle surface.

Function returns
void

There are three arguments to DEFINE_DPM_VP_EQUILIB: name, p, and cvap_surf. You supply the name of your user-defined function. p is passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to compute the equilibrium vapor concentrations and store the values in cvap_surf.

Example

The following UDF named raoult_vpe computes the equilibrium vapor concentration of a multicomponent particle using the Raoult law. The vapor pressure in the law is proportional to the molar fraction of the condenses material. DEFINE_VP_EQUILIB is called several times every particle time step in ANSYS FLUENT and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

```
/***************************************************************************
   UDF for defining the vapor particle equilibrium
   for multicomponent particles
***************************************************************************/
#include <udf.h>
DEFINE_DPM_VP_EQUILIB(raoult_vpe,p,cvap_surf,Z)
{
  int is;
  real molwt[MAX_SPE_EQNS];

  Thread *t0 = P_CELL_THREAD(p);   /* cell thread of particle location */
  Material *gas_mix = THREAD_MATERIAL(t0);  /* gas mixture material */
  Material *cond_mix = P_MATERIAL(p); /* particle mixture material */
  int nc = TP_N_COMPONENTS(p);                /* number of particle components */
  real Tp = P_T(p);                           /* particle temperature */
  real molwt_cond = 0.;      /* reciprocal molecular weight of the particle */

  for (is = 0; is < nc; is++)
    {
      int gas_index = TP_COMPONENT_INDEX_I(p,is); /* index of vaporizing
       component in the gas phase */
      if (gas_index >= 0)
        {
          /* the molecular weight of particle material */
          molwt[gas_index] =
  MATERIAL_PROP(MIXTURE_COMPONENT(gas_mix,gas_index),PROP_mwi);
          molwt_cond += TP_COMPONENT_I(p,is) / molwt[gas_index];
        }
    }

  /* prevent division by zero */
  molwt_cond = MAX(molwt_cond,DPM_SMALL);

  for (is = 0; is < nc; is++)
    {
      /* gas species index of vaporization */
      int gas_index = TP_COMPONENT_INDEX_I(p,is);
      if( gas_index >= 0 )
        {
          /* condensed material */
          Material * cond_c = MIXTURE_COMPONENT( cond_mix, is );
          /* condensed component molefraction */
```

```
        real xi_cond = TP_COMPONENT_I(p,is)/(molwt[gas_index]*molwt_cond);

        /* particle saturation pressure */
        real p_saturation = DPM_vapor_pressure(p, cond_c, Tp);
        if (p_saturation < 0.0)
          p_saturation = 0.0;

        /* vapor pressure over the surface, this is the actual Raoult law */
        cvap_surf[is] = xi_cond * p_saturation / UNIVERSAL_GAS_CONSTANT / Tp;
      }
    }

  /* compressibility for ideal gas */
  *Z = 1.0;
}
```

Hooking a DPM Vapor Equilibrium UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DPM_VP_EQUILIBRIUM is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the Create/Edit Materials dialog box in ANSYS FLUENT. Note that before you hook the UDF, you'll need to create particle injections in the Injections dialog box with the type Multicomponent chosen. See Section 6.4.15: Hooking DEFINE_DPM_VP_EQUILIB UDFs for details on how to hook your DEFINE_DPM_VP_EQUILIB UDF to ANSYS FLUENT.

2.6 Dynamic Mesh DEFINE **Macros**

This section contains descriptions of DEFINE macros that you can use to define UDFs that control the behavior of a dynamic mesh. Note that dynamic mesh UDFs that are defined using DEFINE_CG_MOTION, DEFINE_DYNAMIC_ZONE_PROPERTY, DEFINE_GEOM, and DEFINE_GRID_MOTION can *only* be executed as compiled UDFs.

Table 2.6.1 provides a quick reference guide to the dynamic mesh DEFINE macros, the functions they define, and the dialog boxes where they are activated in ANSYS FLU-ENT. Definitions of each DEFINE macro are contained in the udf.h header file. For your convenience, they are listed in Appendix B.

- Section 2.6.1: DEFINE_CG_MOTION

- Section 2.6.2: DEFINE_DYNAMIC_ZONE_PROPERTY

- Section 2.6.3: DEFINE_GEOM

- Section 2.6.4: DEFINE_GRID_MOTION

- Section 2.6.5: DEFINE_SDOF_PROPERTIES

Table 2.6.1: Quick Reference Guide for Dynamic Mesh-Specific DEFINE Macros

Function	DEFINE Macro	Dialog Box Activated In
center of gravity motion	DEFINE_CG_MOTION	Dynamic Mesh Zones
swirl center	DEFINE_DYNAMIC_ZONE_PROPERTY	In-Cylinder Output Controls
varying cell layering height	DEFINE_DYNAMIC_ZONE_PROPERTY	Dynamic Mesh Zones
mesh motion	DEFINE_GRID_MOTION	Dynamic Mesh Zones
geometry deformation	DEFINE_GEOM	Dynamic Mesh Zones
properties for Six Degrees of Freedom (SDOF) Solver	DEFINE_SDOF_PROPERTIES	Dynamic Mesh Zones

2.6.1 DEFINE_CG_MOTION

Description

You can use DEFINE_CG_MOTION to specify the motion of a particular dynamic zone in ANSYS FLUENT by providing ANSYS FLUENT with the linear and angular velocities at every time step. ANSYS FLUENT uses these velocities to update the node positions on the dynamic zone based on solid-body motion. Note that UDFs that are defined using DEFINE_CG_MOTION can *only* be executed as compiled UDFs.

Usage

DEFINE_CG_MOTION(name,dt,vel,omega,time,dtime)

Argument Type	Description
symbol name	UDF name.
Dynamic_Thread *dt	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by ANSYS FLUENT).
real vel[]	Linear velocity.
real omega[]	Angular velocity.
real time	Current time.
real dtime	Time step.

Function returns
void

There are six arguments to DEFINE_CG_MOTION: name, dt, vel, omega, time, and dtime. You supply name, the name of the UDF. dt, vel, omega, time, and dtime are variables that are passed by the ANSYS FLUENT solver to your UDF. The linear and angular velocities are returned to ANSYS FLUENT by overwriting the arrays vel and omega, respectively.

Example

Consider the following example where the linear velocity is computed from a simple force balance on the body in the x-direction such that

$$\int_{t_o}^{t} dv = \int_{t_o}^{t} (F/m) \, dt \qquad (2.6\text{-}1)$$

where v is velocity, F is the force and m is the mass of the body. The velocity at time t is calculated using an explicit Euler formula as

$$v_t = v_{t-\Delta t} + (F/m)\Delta t \qquad (2.6\text{-}2)$$

```
/****************************************************************
 * 1-degree of freedom equation of motion (x-direction)
 * compiled UDF
 ****************************************************************/
#include "udf.h"
static real v_prev = 0.0;

DEFINE_CG_MOTION(piston,dt,vel,omega,time,dtime)
{
  Thread *t;
  face_t f;
  real NV_VEC(A);
  real force, dv;

  /* reset velocities */
  NV_S(vel, =, 0.0);
  NV_S(omega, =, 0.0);

  if (!Data_Valid_P())
    return;

  /* get the thread pointer for which this motion is defined */
  t = DT_THREAD(dt);

  /* compute pressure force on body by looping through all faces */
  force = 0.0;
  begin_f_loop(f,t)
    {
      F_AREA(A,f,t);
      force += F_P(f,t) * NV_MAG(A);
```

```
  }
  end_f_loop(f,t)

  /* compute change in velocity, i.e., dv = F * dt / mass
     velocity update using explicit Euler formula */
  dv = dtime * force / 50.0;
  v_prev += dv;
  Message ("time = %f, x_vel = %f, force = %f\n", time, v_prev,
  force);

  /* set x-component of velocity */
  vel[0] = v_prev;
}
```

Hooking a Center of Gravity Motion UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_CG_MOTION is compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Dynamic Mesh Zones dialog box in ANSYS FLUENT. See Section 6.5.1: Hooking DEFINE_CG_MOTION UDFs for details on how to hook your DEFINE_CG_MOTION UDF to ANSYS FLUENT.

2.6.2 DEFINE_DYNAMIC_ZONE_PROPERTY

Description

The DEFINE_DYNAMIC_ZONE_PROPERTY UDF can be used in the following applications:

- swirl center definition for in-cylinder applications

- variable cell layering height

Swirl Center Definition for In-Cylinder Applications

You can use DEFINE_DYNAMIC_ZONE_PROPERTY to calculate swirl center while computing in-cylinder specific output.

i Note that UDFs that are defined using DEFINE_DYNAMIC_ZONE_PROPERTY can *only* be executed as compiled UDFs.

For information on setting in-cylinder parameters, see Section 11.3.6: Setting In-Cylinder Parameters in the separate User's Guide.

Usage

```
DEFINE_DYNAMIC_ZONE_PROPERTY(name, dt, swirl_center)
```

Argument Type	Description
symbol name	UDF name.
Dynamic_Thread *dt	Pointer to a structure that stores the dynamic mesh attributes. This is set to NULL internally as there are no dynamic zones in the current calculation of swirl center.
real *swirl_center	Pointer to a real array of 3 dimension. You will assign this value in the UDF. The x, y and z values of the swirl_center can be assigned in the UDF through swirl_center[0], swirl_center[1] and swirl_center[2] respectively.

Function returns
```
void
```

There are three arguments to DEFINE_DYNAMIC_ZONE_PROPERTY: name, dt, and swirl_center. You supply name, the name of the UDF, and pointer to a real array, swirl_center. dt is a variable that is passed by the ANSYS FLUENT solver to your UDF.

Example

```
/* UDF hook for calculating Swirl Center while computing
    In-Cylinder specific output.  Arguments for the UDF
    hook are name of the UDF, dt (dynamic thread) which is
    set to NULL and it is not supposed to be manipulated
    in the UDF, as there are no dynamic zones in the current
    context and swirl center which is to be calculated in the
    UDF.  Works in parallel as well.
 */

#include "udf.h"
#define RPM RP_Get_Real("dynamesh/in-cyn/crank-rpm")

static real Zmin_at_TDC = -0.0014; /* Piston location at TDC */
static real Zmax = 0.0145;         /* Zmax, a fixed point */

static void my_swirl_center(real * swirl_center)
{
```

```
  real piston_displacement, lambda, CA, l, r;
#if !RP_NODE
  l = RP_Get_List_Ref_Float("dynamesh/in-cyn/piston-data", 0);
  r= 0.5 * RP_Get_List_Ref_Float("dynamesh/in-cyn/piston-data",1);
#endif

  host_to_node_real_2(l,r);
  lambda = r/l;

  CA = (CURRENT_TIME*RPM*6.0 +
        RP_Get_Real("dynamesh/in-cyn/crank-start-angle"))*M_PI/180;
  piston_displacement = r*((1+1/lambda) - cos(CA) -
                           pow(1-lambda*lambda*sin(CA)*sin(CA),0.5)/lambda);

  swirl_center[0]=0;
  swirl_center[1]=0;
  if (Zmin_at_TDC<Zmax)
    swirl_center[2]=0.5*(Zmin_at_TDC+Zmax-piston_displacement);
  else
    swirl_center[2]=0.5*(Zmin_at_TDC+Zmax+piston_displacement);

  return;
}


DEFINE_DYNAMIC_ZONE_PROPERTY(swirl_udf, dt, sc)
{
  my_swirl_center(sc);
}
```

Hooking a Swirl Center UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DYNAMIC_ZONE_PROPERTY is compiled (as described in Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the In-Cylinder Output Controls dialog box in ANSYS FLUENT.

See Section 6.5.2: Hooking DEFINE_DYNAMIC_ZONE_PROPERTY UDFs for details on how to hook your DEFINE_DYNAMIC_ZONE_PROPERTY UDF to ANSYS FLUENT.

Variable Cell Layering Height

You can use DEFINE_DYNAMIC_ZONE_PROPERTY to specify a varying cell layering height when using the dynamic layering method to split or merge cells adjacent to a moving boundary. The cell layering height can be specified as a function of time for general applications, or as a function of crank angle for in-cylinder applications.

i Note that UDFs that are defined using DEFINE_DYNAMIC_ZONE_PROPERTY can *only* be executed as compiled UDFs.

For information on the dynamic layering method, see Section 11.3.2: Dynamic Layering Method in the separate User's Guide.

Usage

DEFINE_DYNAMIC_ZONE_PROPERTY(name, dt, height)

Argument Type	Description
symbol name	UDF name.
Dynamic_Thread *dt	Pointer to a structure that stores the dynamic mesh attributes.
real *height	Pointer to a real value layering height whose value will be varied in the UDF as a function of time or crank angle.

Function returns
void

There are three arguments to DEFINE_DYNAMIC_ZONE_PROPERTY: name, dt, and height. You supply name, the name of the UDF, and height, the cell layering height to be assigned in the UDF as a function of time / crank angle. dt is a variable that is passed by the ANSYS FLUENT solver to your UDF.

In addition to the arguments listed previously, you can utilize the variable in_cyl_ca_period and the macros DYNAMESH_CURRENT_TIME and TIME_TO_ABSOLUTE_CRANK_ANGLE(time), which are described as follows:

Variable/Macro	Description
in_cyl_ca_period	Crank angle period.
DYNAMESH_CURRENT_TIME	Current dynamic mesh time.
TIME_TO_ABSOLUTE_CRANK_ANGLE(time)	Macro which takes the current time as input and returns the absolute value of the crank angle that is displayed on the mesh preview screen.

Note that in_cyl_ca_period is the value entered for Crank Period in the In-Cylinder Settings dialog box (which can be opened via the Dynamic Mesh task page). The usage of this variable or the macros specified previously necessitates that the DEFINE_DYNAMIC_ZONE_PROPERTY UDF be a compiled UDF. Their usage is illustrated in the example that follows.

Note that the header file dynamesh_tools.h should be included in the UDF, as shown in the example that follows.

Example

```
/* UDF hook for implementing varying cell layering height.
    Arguments are the Name of the UDF,
    variable for dynamic thread, and variable
    which holds the layering height value.
    Works only as a compiled UDF, because the usage of
    in_cyn_ca_period and the macros are not
    allowed in interpreted UDFs.
    Header file dynamesh_tools.h should be
    included in order to access the macros
    DYNAMESH_CURRENT_TIME and TIME_TO_ABSOLUTE_CRANK_ANGLE
*/

#include "udf.h"
#include "dynamesh_tools.h"

DEFINE_DYNAMIC_ZONE_PROPERTY(nonconst_height, dt, lh )
{
int temp;

/* Local variable for storing the value of
   Absolute Crank Angle */ real abs_ca;

/* Local variables for saving time and
   Crank Angle, etc. */ real half,quart,time,ca;

half = in_cyn_ca_period / 2.0;
quart = in_cyn_ca_period /4.0;

time = DYNAMESH_CURRENT_TIME;

ca = TIME_TO_ABSOLUTE_CRANK_ANGLE(time);
temp = (int) ( ca / half);
abs_ca = ca - temp * half ;
```

```
/* *lh controls the layering height */
if( abs_ca <= quart )
*lh = ( 0.5 + (abs_ca)/ quart * 0.8);
else
*lh = ( 0.5 + ( (half - abs_ca) / quart ) * 0.8);


}
```

Hooking a Variable Cell Layering Height UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_DYNAMIC_ZONE_PROPERTY is compiled (as described in Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Dynamic Mesh Zones dialog box in ANSYS FLUENT.

See Section 6.5.2: Hooking DEFINE_DYNAMIC_ZONE_PROPERTY UDFs for details on how to hook your DEFINE_DYNAMIC_ZONE_PROPERTY UDF to ANSYS FLUENT.

2.6.3 DEFINE_GEOM

Description

You can use DEFINE_GEOM to specify the geometry of a deforming zone. By default, ANSYS FLUENT provides a mechanism for defining node motion along a planar or cylindrical surface. When ANSYS FLUENT updates a node on a deforming zone (e.g., through spring-based smoothing or after local face re-meshing) the node is "repositioned" by calling the DEFINE_GEOM UDF. Note that UDFs that are defined using DEFINE_GEOM can *only* be executed as compiled UDFs.

Usage

DEFINE_GEOM(name,d,dt,position)

Argument Type	Description
symbol name	UDF name.
Domain *d	Pointer to domain.
Dynamic_Thread *dt	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by ANSYS FLUENT).
real *position	Pointer to array that stores the position.

Function returns
void

There are four arguments to DEFINE_GEOM: name, d, dt, and position. You supply name, the name of the UDF. d, dt, and position are variables that are passed by the ANSYS FLUENT solver to your UDF. The new position (after projection to the geometry defining the zone) is returned to ANSYS FLUENT by overwriting the position array.

Example

The following UDF, named parabola, is executed as a compiled UDF.

```
/***********************************************************
 * defining parabola through points (0, 1), (1/2, 5/4), (1, 1)
 ***********************************************************/
#include "udf.h"

DEFINE_GEOM(parabola,domain,dt,position)
{
  /* set y = -x^2 + x + 1 */
  position[1] = - position[0]*position[0] + position[0] + 1;
}
```

Hooking a Dynamic Mesh Geometry UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_GEOM is compiled (see Chapter 5: Compiling UDFs for details), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Dynamic Mesh Zones dialog box in ANSYS FLUENT. See Section 6.5.3: Hooking DEFINE_GEOM UDFs for details on how to hook your DEFINE_GEOM UDF to ANSYS FLUENT.

2.6.4 DEFINE_GRID_MOTION

Description

By default, ANSYS FLUENT updates the node positions on a dynamic zone by applying the solid-body motion equation. This implies that there is no relative motion between the nodes on the dynamic zone. However, if you need to control the motion of each node independently, then you can use DEFINE_GRID_MOTION UDF. A mesh motion UDF can, for example, update the position of each node based on the deflection due to fluid-structure interaction. Note that UDFs that are defined using DEFINE_GRID_MOTION can be executed *only* as compiled UDFs.

Usage

DEFINE_GRID_MOTION(name, d, dt, time, dtime)

Argument Type	Description
symbol name	UDF name.
Domain *d	Pointer to domain.
Dynamic_Thread *dt	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by ANSYS FLUENT).
real time	Current time.
real dtime	Time step.

Function returns
void

There are five arguments to DEFINE_GRID_MOTION: name, d, dt, time, and dtime. You supply name, the name of the UDF. d, dt, time, and dtime are variables that are passed by the ANSYS FLUENT solver to your UDF.

Example

Consider the following example where you want to specify the deflection on a cantilever beam based on the x position such that

$$
\begin{aligned}
\omega_y(x) &= -10.4\sqrt{x}\sin 26.178\ t \qquad x > 0.02 & (2.6\text{-}3)\\
\omega_y(x) &= 0 \qquad x <= 0.02 & (2.6\text{-}4)
\end{aligned}
$$

where $\omega_y(x)$ is the y-component of the angular velocity at a position x. The node position is updated based on

$$
(\vec{r})^{t+\Delta t} = (\vec{r})^t + \vec{\Omega} \times (\vec{r})^t \Delta t \qquad (2.6\text{-}5)
$$

where $\vec{\Omega}$ is the angular velocity and $\vec{r}$ is the position vector of a node on the dynamic zone.

```
/**********************************************************
 node motion based on simple beam deflection equation
 compiled UDF
 **********************************************************/
#include "udf.h"

DEFINE_GRID_MOTION(beam,domain,dt,time,dtime)
{
  Thread *tf = DT_THREAD(dt);
  face_t f;
  Node *v;
  real NV_VEC(omega), NV_VEC(axis), NV_VEC(dx);
  real NV_VEC(origin), NV_VEC(rvec);
  real sign;
  int n;

  /* set deforming flag on adjacent cell zone */
  SET_DEFORMING_THREAD_FLAG(THREAD_T0(tf));

  sign = -5.0 * sin (26.178 * time);

  Message ("time = %f, omega = %f\n", time, sign);

  NV_S(omega, =, 0.0);
  NV_D(axis, =, 0.0, 1.0, 0.0);
```

```
   NV_D(origin, =, 0.0, 0.0, 0.152);

   begin_f_loop(f,tf)
     {
       f_node_loop(f,tf,n)
         {
           v = F_NODE(f,tf,n);

           /* update node if x position is greater than 0.02
              and that the current node has not been previously
              visited when looping through previous faces */
           if (NODE_X(v) > 0.020 && NODE_POS_NEED_UPDATE (v))
             {
               /* indicate that node position has been update
                  so that it's not updated more than once */
               NODE_POS_UPDATED(v);

               omega[1] = sign * pow (NODE_X(v)/0.230, 0.5);
               NV_VV(rvec, =, NODE_COORD(v), -, origin);
               NV_CROSS(dx, omega, rvec);
               NV_S(dx, *=, dtime);
               NV_V(NODE_COORD(v), +=, dx);
             }
         }
     }

   end_f_loop(f,tf);
}
```

Hooking a DEFINE_GRID_MOTION **to** ANSYS FLUENT

After the UDF that you have defined using DEFINE_GRID_MOTION is compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Dynamic Mesh Zones dialog box in ANSYS FLUENT. See Section 6.5.4: Hooking DEFINE_GRID_MOTION UDFs for details on how to hook your DEFINE_GRID_MOTION UDF to ANSYS FLUENT.

2.6.5 DEFINE_SDOF_PROPERTIES

Description

You can use DEFINE_SDOF_PROPERTIES to specify custom properties of moving objects for the six-degrees of freedom (SDOF) solver in **ANSYS FLUENT**. These include mass, moment and products of inertia, and external forces and moment properties. The properties of an object which can consist of multiple zones can change in time, if desired. External load forces and moments can either be specified as global coordinates or body coordinates. In addition, you can specify custom transformation matrices using DEFINE_SDOF_PROPERTIES.

Usage

DEFINE_SDOF_PROPERTIES(name,properties,dt,time,dtime)

Argument Type	Description
symbol name	UDF name.
real *properties	Pointer to the array that stores the SDOF properties.
Dynamic_Thread *dt	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by **ANSYS FLUENT**).
real time	Current time.
real dtime	Time step.

Function returns
void

There are four arguments to DEFINE_SDOF_PROPERTIES: name, properties, dt, and dtime. You provide the name of the UDF. properties, dt, and dtime are variables that are passed by the **ANSYS FLUENT** solver to your UDF. The property array pointer that is passed to your function allows you to specify values for any of the following SDOF properties:

```
SDOF_MASS          /* mass */
SDOF_IXX,          /* moment of inertia */
SDOF_IYY,          /* moment of inertia */
SDOF_IZZ,          /* moment of inertia */
SDOF_IXY,          /* product of inertia */
SDOF_IXZ,          /* product of inertia */
SDOF_IYZ,          /* product of inertia */
SDOF_LOAD_LOCAL,   /* boolean */
SDOF_LOAD_F_X,     /* external force */
```

```
   SDOF_LOAD_F_Y,      /* external force */
   SDOF_LOAD_F_Z,      /* external force */
   SDOF_LOAD_M_X,      /* external moment */
   SDOF_LOAD_M_Y,      /* external moment */
   SDOF_LOAD_M_Z,      /* external moment */
```

The boolean `prop[SDOF_LOAD_LOCAL]` can be used to determine whether the forces and moments are expressed in terms of global coordinates (`FALSE`) or body coordinates (`TRUE`). The default value for `prop[SDOF_LOAD_LOCAL]` is `FALSE`.

Custom Transformation Variables

The default transformations used by ANSYS FLUENT are typical for most aerospace and other types of applications. However, if your model requires custom transformations, you can specify these matrices in your SDOF UDF. First set the `SDOF_CUSTOM_TRANS` boolean to `TRUE`. Then use the macros listed below to define custom coordination rotation and derivative rotation matrices. `CTRANS` is the body-global coordinate rotation matrix and `DTRANS` is the body-global derivative rotation matrix.

```
   SDOF_CUSTOM_TRANS,       /* boolean */
   SDOF_CTRANS_11,          /* coordinate rotation matrices */
   SDOF_CTRANS_12,
   SDOF_CTRANS_13,
   SDOF_CTRANS_21,
   SDOF_CTRANS_22,
   SDOF_CTRANS_23,
   SDOF_CTRANS_31,
   SDOF_CTRANS_32,
   SDOF_CTRANS_33,
   SDOF_DTRANS_11,          /* derivative rotation matrices */
   SDOF_DTRANS_12,
   SDOF_DTRANS_13,
   SDOF_DTRANS_21,
   SDOF_DTRANS_22,
   SDOF_DTRANS_23,
   SDOF_DTRANS_31,
   SDOF_DTRANS_32,
   SDOF_DTRANS_33,
```

Example 1

The following UDF, named `stage`, is a simple example of setting mass and moments of inertia properties for a moving object. This UDF is typical for applications in which a body is dropped and the SDOF solver computes the body's motion in the flow field.

```
/************************************************************
 Simple example of a SDOF property UDF for a moving body
*************************************************************/
#include "udf.h"

DEFINE_SDOF_PROPERTIES(stage, prop, dt, time, dtime)
{
   prop[SDOF_MASS]        = 800.0;
   prop[SDOF_IXX]         = 200.0;
   prop[SDOF_IYY]         = 100.0;
   prop[SDOF_IZZ]         = 100.0;

   printf ("\nstage: updated 6DOF properties");
}
```

Example 2

The following UDF named `delta_missile` specifies case injector forces and moments that are time-dependent. Specifically, the external forces and moments depend on the current angular orientation of the moving object. Note that this UDF must be executed as a compiled UDF.

```
/*********************************************************
SDOF property compiled UDF with external forces/moments
**********************************************************/
#include "udf.h"

DEFINE_SDOF_PROPERTIES(delta_missile, prop, dt, time, dtime)
{
   prop[SDOF_MASS]        = 907.185;
   prop[SDOF_IXX]         = 27.116;
   prop[SDOF_IYY]         = 488.094;
   prop[SDOF_IZZ]         = 488.094;

   /* add injector forces, moments */
   {
     register real dfront = fabs (DT_CG (dt)[2] -
```

```
                                (0.179832*DT_THETA (dt)[1]));
     register real dback  = fabs (DT_CG (dt)[2] +
                                (0.329184*DT_THETA (dt)[1]));

     if (dfront <= 0.100584)
       {
         prop[SDOF_LOAD_F_Z] = 10676.0;
         prop[SDOF_LOAD_M_Y] = -1920.0;
       }

     if (dback <= 0.100584)
       {
         prop[SDOF_LOAD_F_Z] += 42703.0;
         prop[SDOF_LOAD_M_Y] += 14057.0;
       }
   }

   printf ("\ndelta_missile: updated 6DOF properties");
}
```

Hooking a DEFINE_SDOF_PROPERTIES **UDF to** ANSYS FLUENT

After the UDF that you have defined using DEFINE_SDOF_PROPERTIES is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the Six DOF UDF drop-down list in the Dynamic Mesh Zones dialog box in ANSYS FLUENT. See Section 6.5.5: Hooking DEFINE_SDOF_PROPERTIES UDFs for details on how to hook your DEFINE_SDOF_PROPERTIES UDF to ANSYS FLUENT.

2.7   User-Defined Scalar (UDS) Transport Equation `DEFINE` Macros

This section provides information on how you can define UDFs that can be used in UDS transport equations in ANSYS FLUENT. See Section 9.1: User-Defined Scalar (UDS) Transport Equations in the separate User's Guide for UDS equation theory and details on how to setup scalar equations. Descriptions of `DEFINE` macros for UDS applications are provided below. Definitions of `DEFINE` macros are contained in the `udf.h` header file. For your convenience, they are also listed in Appendix B. Detailed examples of user-defined scalar transport UDFs can be found in Section 8.2.5: User-Defined Scalars.

- Section 2.7.1: Introduction

- Section 2.7.2: `DEFINE_ANISOTROPIC_DIFFUSIVITY`

- Section 2.7.3: `DEFINE_UDS_FLUX`

- Section 2.7.4: `DEFINE_UDS_UNSTEADY`

2.7.1   Introduction

For each of the N scalar equations you specified in your ANSYS FLUENT model you can supply a unique UDF for the diffusion coefficients, flux, and unsteady terms in the scalar transport equation. For multiphase you have the added benefit of specifying UDFs on a per-phase basis in both fluid and solid zones. Additionally, you can specify a UDF for each source term you define for a given scalar equation as well as boundary conditions on wall, inflow, and outflow boundaries.

Diffusion Coefficient UDFs

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can supply a unique user-defined function (UDF) for isotropic and anisotropic diffusivity for both fluid and solid materials. Recall that ANSYS FLUENT computes the diffusion coefficient in the UDS equation.

Isotropic diffusivity UDFs are defined using the `DEFINE_DIFFUSIVITY` macro (Section 2.3.3: `DEFINE_DIFFUSIVITY`) and anisotropic coefficients UDFs are defined using `DEFINE_ANISOTROPIC_DIFFFUSIVITY` (Section 2.7.2: `DEFINE_ANISOTROPIC_DIFFUSIVITY`). Additional pre-defined macros that you can use when coding UDS functions are provided in Section 3.2.8: User-Defined Scalar (UDS) Transport Equation Macros.

Flux UDFs

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can supply a unique user-defined function (or UDF) for the advective flux term. Recall that ANSYS FLUENT computes the flux in the UDS equation.

UDS Flux UDFs are defined using the DEFINE_UDS_FLUX macro (Section 2.7.3: DEFINE_UDS_FLUX). Additional pre-defined macros that you can use when coding scalar flux UDFs are provided in Section 3.2.8: User-Defined Scalar (UDS) Transport Equation Macros.

Unsteady UDFs

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can supply a unique UDF for the unsteady function. Recall that ANSYS FLUENT computes the unsteady term in the UDS equation.

Scalar Unsteady UDFs are defined using the DEFINE_UDS_UNSTEADY macro (Section 2.7.4: DEFINE_UDS_UNSTEADY). Additional pre-defined macros that you can use when coding scalar unsteady UDFs are provided in Section 3.2.8: User-Defined Scalar (UDS) Transport Equation Macros.

Source Term UDFs

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can supply a unique UDF for *each* source. Recall that ANSYS FLUENT computes the source term in the UDS equation.

Scalar source UDFs are defined using the DEFINE_SOURCE macro and must compute the source term, S_{ϕ_k}, and its derivative $\frac{\partial S_{\phi_k}}{\partial \phi_k}$ (Section 2.3.19: DEFINE_SOURCE). Additional pre-defined macros that you can use when coding scalar source term UDFs are provided in Section 3.2.8: User-Defined Scalar (UDS) Transport Equation Macros.

Fixed Value Boundary Condition UDFs

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can supply a fixed value profile UDF for fluid boundaries.

Fixed value UDFs are defined using the DEFINE_PROFILE macro. See Section 2.3.15: DEFINE_PROFILE for details. Additional pre-defined macros that you can use for coding scalar transport equation UDFs are provided in Section 3.2.8: User-Defined Scalar (UDS) Transport Equation Macros.

Wall, Inflow, and Outflow Boundary Condition UDFs

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can supply a specified value or flux UDF for all wall, inflow, and outflow boundaries.

Wall, inflow, and outflow boundary UDFs are defined using the DEFINE_PROFILE macro (Section 2.3.15: DEFINE_PROFILE). Additional pre-defined macros that you can use for coding scalar transport equation UDFs are provided in Section 3.2.8: User-Defined Scalar (UDS) Transport Equation Macros.

2.7.2 DEFINE_ANISOTROPIC_DIFFUSIVITY

Description

You can use DEFINE_ANISOTROPIC_DIFFUSIVITY to specify an anisotropic diffusivity for a user-defined scalar (UDS) transport equation. See Section 8.6.2: Anisotropic Diffusion in the separate User's Guide for details about anisotropic diffusivity material properties in ANSYS FLUENT.

Usage

DEFINE_ANISOTROPIC_DIFFUSIVITY(name,c,t,i,dmatrix)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the anisotropic diffusivity function is to be applied.
int i	Index that identifies the user-defined scalar.
real dmatrix[ND_ND][ND_ND]	Anisotropic diffusivity matrix to be filled in by user.

Function returns
void

There are five arguments to DEFINE_ANISOTROPIC_DIFFUSIVITY: name, c, t, i, and dmatrix. You will supply name, the name of the UDF. c, t, i, and dmatrix are variables that are passed by the ANSYS FLUENT solver to your UDF. Your function will compute the diffusivity tensor for a single cell and fill dmatrix with it. Note that anisotropic diffusivity UDFs are called by ANSYS FLUENT from within a loop on cell threads. Consequently, your UDF will not need to loop over cells in a thread since ANSYS FLUENT is doing it outside of the function call.

Example

The following UDF, named `cyl_ortho_diff` computes the anisotropic diffusivity matrix for a cylindrical shell which has different diffusivities in radial, tangential, and axial directions. This function can be executed as a compiled UDF.

```
/******************************************************************
   Example UDF that demonstrates DEFINE_ANISOTROPIC_DIFFUSIVITY
******************************************************************/
#include "udf.h"

/* Computation of anisotropic diffusivity matrix for
 * cylindrical orthotropic diffusivity */

/* axis definition for cylindrical diffusivity */
static const real origin[3] = {0.0, 0.0, 0.0};
static const real axis[3]   = {0.0, 0.0, 1.0};

/* diffusivities in radial, tangential and axial directions */
static const real diff[3] = {1.0, 0.01, 0.01};

DEFINE_ANISOTROPIC_DIFFUSIVITY(cyl_ortho_diff,c,t,i,dmatrix)
{
    real x[3][3]; /* principal direction matrix for cell in
cartesian coords. */
    real xcent[ND_ND];
    real R;

    C_CENTROID(xcent,c,t);

    NV_VV(x[0],=,xcent,-,origin);
#if RP_3D
    NV_V(x[2],=,axis);
#endif
#if RP_3D
    R = NV_DOT(x[0],x[2]);
    NV_VS(x[0],-=,x[2],*,R);
#endif
    R = NV_MAG(x[0]);
    if (R > 0.0)
      NV_S(x[0],/=,R);
#if RP_3D
    N3V_CROSS(x[1],x[2],x[0]);
#else
```

```
    x[1][0] = -x[0][1];
    x[1][1] =  x[0][0];
#endif

    /* dmatrix is computed as xT*diff*x */
    dmatrix[0][0] = diff[0]*x[0][0]*x[0][0]
      + diff[1]*x[1][0]*x[1][0]
#if RP_3D
      + diff[2]*x[2][0]*x[2][0]
#endif
      ;
    dmatrix[1][1] = diff[0]*x[0][1]*x[0][1]
      + diff[1]*x[1][1]*x[1][1]
#if RP_3D
      + diff[2]*x[2][1]*x[2][1]
#endif
      ;
    dmatrix[1][0] = diff[0]*x[0][1]*x[0][0]
      + diff[1]*x[1][1]*x[1][0]
#if RP_3D
      + diff[2]*x[2][1]*x[2][0]
#endif
      ;
    dmatrix[0][1] = dmatrix[1][0];

#if RP_3D
    dmatrix[2][2] = diff[0]*x[0][2]*x[0][2]
      + diff[1]*x[1][2]*x[1][2]
      + diff[2]*x[2][2]*x[2][2]
      ;
    dmatrix[0][2] = diff[0]*x[0][0]*x[0][2]
      + diff[1]*x[1][0]*x[1][2]
      + diff[2]*x[2][0]*x[2][2]
      ;
    dmatrix[2][0] = dmatrix[0][2];

    dmatrix[1][2] = diff[0]*x[0][1]*x[0][2]
      + diff[1]*x[1][1]*x[1][2]
      + diff[2]*x[2][1]*x[2][2]
      ;
    dmatrix[2][1] = dmatrix[1][2];
#endif
}
```

Hooking an Anisotropic Diffusivity UDF to ANSYS FLUENT

After the UDF that you have defined using DEFINE_ANISOTROPIC_DIFFUSIVITY is in-
terpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs),
the name of the argument that you supplied as the first DEFINE macro argument (e.g.,
cyl_ortho_diff) will become selectable via the UDS Diffusion Coefficients dialog box.
You'll first need to select defined-per-uds for UDS Diffusivity in the Create/Edit Materials
dialog box, then select the user-defined-anisotropic option for Coefficient from the
UDS Diffusion Coefficients dialog box for a particular user-defined scalar diffusion equa-
tion (e.g., uds-0). See Section 6.6.1: Hooking DEFINE_ANISOTROPIC_DIFFUSIVITY UDFs
for details.

2.7.3 DEFINE_UDS_FLUX

Description

You can use DEFINE_UDS_FLUX to customize how the advective flux term is computed in
your user-defined scalar (UDS) transport equations. See Section 9.1: User-Defined Scalar
(UDS) Transport Equations in the separate User's Guide for details on setting up and
solving UDS transport equations.

Usage

DEFINE_UDS_FLUX(name,f,t,i)

Argument Type	Description
symbol name	UDF name.
face_t f	Face index.
Thread *t	Pointer to face thread on which the user-defined scalar flux is to be applied.
int i	Index that identifies the user-defined scalar for which the flux term is to be set.

Function returns
real

There are four arguments to DEFINE_UDS_FLUX: name, f, t, and i. You supply name, the
name of the UDF. f, t, and i are variables that are passed by the ANSYS FLUENT solver
to your UDF. Your UDF will need to return the real value of the mass flow rate through
the given face to the solver.

The advection term in the differential transport equation has the following most general form:

$$\nabla \cdot \vec{\psi} \phi \qquad (2.7\text{-}1)$$

where ϕ is the user-defined scalar conservation quantity and $\vec{\psi}$ is a vector field. In the default advection term, $\vec{\psi}$ is, by default, the product of the scalar density and the velocity vector:

$$\vec{\psi}_{\text{default}} = \rho \vec{v} \qquad (2.7\text{-}2)$$

To define the advection term in Equation 2.7-1 using `DEFINE_UDS_FLUX`, your UDF needs to return the scalar value $\vec{\psi} \cdot \vec{A}$ to ANSYS FLUENT, where $\vec{\psi}$ is the same as defined in Equation 2.7-1 and $\vec{A}$ is the face normal vector of the face.

> i Note that the advective flux field that is supplied by your UDF should be divergence-free (i.e., it satisfies the continuity equation). In discrete terms this means that the sum of fluxes over all the faces of each cell should be zero. If the advective field is not divergence-free, then ϕ is not "conserved" and will result in overshoots/undershoots in the cell value of ϕ.

You will need to compute $\vec{\psi}$ in your UDF using, for example, predefined macros for velocity vector and scalar density that ANSYS FLUENT has provided (see Chapter 3: Additional Macros for Writing UDFs) or using your own prescription. The first case is illustrated in the sample C source code, shown below.

> i Note that if more than one scalar is being solved, you can use a conditional `if` statement in your UDF to define a different flux function for each `i`. `i` = 0 is associated with scalar-0 (the first scalar equation being solved).

> i Note also that $\vec{\psi} \cdot \vec{A}$ must have units of mass flow rate in SI (i.e., kg/s).

```
/************************************************************************
    sample C source code that computes dot product of psi and A
    Note that this is not a complete C function
 ************************************************************************/

real NV_VEC(psi), NV_VEC(A);          /* declaring vectors psi and A  */

                             /* defining psi in terms of velocity field */
NV_D(psi,  =, F_U(f,t), F_V(f,t), F_W(f,t));

NV_S(psi, *=, F_R(f,t))   /* multiplying density to get psi vector   */

F_AREA(A,f,t)                  /* face normal vector returned from F_AREA */

return NV_DOT(psi,A);      /* dot product of the two returned         */
```

Additionally, since most quantities in ANSYS FLUENT are not allocated in memory for interior faces, only for boundary faces (e.g., wall zones), your UDF will also need to calculate interior face values from the cell values of adjacent cells. This is most easily done using the arithmetic mean method. Vector arithmetic can be coded in C using the NV‿ and ND‿ macros (see Chapter 3: Additional Macros for Writing UDFs).

Note that if you had to implement the default advection term in a UDF without the fluid density in the definition of ψ (see above), you could simply put the following line in your DEFINE‿UDS‿FLUX UDF:

```
return F_FLUX(f,t) / rho;
```

where the denominator ρ can be determined by averaging the adjacent cell's density values C‿R(F‿C0(f,t),THREAD‿T0(t)) and C‿R(F‿C1(f,t),THREAD‿T1(t)).

Example

The following UDF, named my‿uds‿flux, returns the mass flow rate through a given face. The flux is usually available through the ANSYS FLUENT-supplied macro F‿FLUX(f,t) (Section 3.2.4: Face Macros). The sign of flux that is computed by the ANSYS FLUENT solver is positive if the flow direction is the same as the face area normal direction (as determined by F‿AREA - see Section 3.2.4: Face Area Vector (F‿AREA)), and is negative if the flow direction and the face area normal directions are opposite. By convention, face area normals always point out of the domain for boundary faces, and they point in the direction from cell c0 to cell c1 for interior faces.

The UDF must be executed as a compiled UDF.

```
/*********************************************************************/
/*      UDF that implements a simplified advective term in the       */
/*      scalar transport equation                                    */
/*********************************************************************/

#include "udf.h"

DEFINE_UDS_FLUX(my_uds_flux,f,t,i)
{
  cell_t  c0,  c1 = -1;
  Thread *t0, *t1 = NULL;

  real NV_VEC(psi_vec), NV_VEC(A), flux = 0.0;

  c0 = F_C0(f,t);
  t0 = F_C0_THREAD(f,t);
  F_AREA(A, f, t);

  /* If face lies at domain boundary, use face values; */
  /* If face lies IN the domain, use average of adjacent cells. */

 if (BOUNDARY_FACE_THREAD_P(t)) /*Most face values will be available*/
    {
      real dens;

      /* Depending on its BC, density may not be set on face thread*/
      if (NNULLP(THREAD_STORAGE(t,SV_DENSITY)))
        dens = F_R(f,t);   /* Set dens to face value if available */
      else
        dens = C_R(c0,t0); /* else, set dens to cell value */

      NV_DS(psi_vec,  =, F_U(f,t), F_V(f,t), F_W(f,t), *, dens);

      flux = NV_DOT(psi_vec, A); /* flux through Face */
    }
  else
    {
      c1 = F_C1(f,t);          /* Get cell on other side of face */
      t1 = F_C1_THREAD(f,t);

      NV_DS(psi_vec,  =, C_U(c0,t0),C_V(c0,t0),C_W(c0,t0),*,C_R(c0,t0));
      NV_DS(psi_vec, +=, C_U(c1,t1),C_V(c1,t1),C_W(c1,t1),*,C_R(c1,t1));
```

```
        flux = NV_DOT(psi_vec, A)/2.0; /* Average flux through face */
    }

  /* ANSYS FLUENT will multiply the returned value by phi_f (the scalar's
     value at the face) to get the ''complete'' advective term.  */

  return flux;
}
```

Hooking a UDS Flux Function to ANSYS FLUENT

After the UDF that you have defined using DEFINE_UDS_FLUX is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., my_uds_flux) will become visible and selectable in the User-Defined Scalars dialog box in ANSYS FLUENT. See Section 6.6.2: Hooking DEFINE_UDS_FLUX UDFs for details.

2.7.4 DEFINE_UDS_UNSTEADY

Description

You can use DEFINE_UDS_UNSTEADY to customize unsteady terms in your user-defined scalar (UDS) transport equations. See Section 9.1: User-Defined Scalar (UDS) Transport Equations in the separate User's Guide for details on setting up and solving UDS transport equations.

Usage

DEFINE_UDS_UNSTEADY(name,c,t,i,apu,su)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread on which the unsteady term for the user-defined scalar transport equation is to be applied.
int i	Index that identifies the user-defined scalar for which the unsteady term is to be set.
real *apu	Pointer to central coefficient.
real *su	Pointer to source term.

Function returns
void

There are six arguments to DEFINE_UDS_UNSTEADY: name, c, t, i, apu, and su. You supply name, the name of the UDF. c, t, and i are variables that are passed by the ANSYS FLUENT solver to your UDF. Your UDF will need to set the values of the unsteady terms referenced by the real pointers apu and su to the central coefficient and source term, respectively.

The ANSYS FLUENT solver expects that the transient term will be decomposed into a source term, su, and a central coefficient term, apu. These terms are included in the equation set in a similar manner to the way the explicit and implicit components of a source term might be handled. Hence, the unsteady term is moved to the right-hand side and discretized as follows:

$$
\begin{aligned}
\text{unsteady term} \quad &= \quad -\int \frac{\partial}{\partial t}\left(\rho\phi\right) dV \\
&\approx \quad -\left[\frac{(\rho\phi)^n - (\rho\phi)^{n-1}}{\Delta t}\right] \cdot \Delta V \\
&= \quad \underbrace{-\frac{\rho\Delta V}{\Delta t}}_{\text{apu}} \phi^n + \underbrace{\frac{\rho\Delta V}{\Delta t}}_{\text{su}} \phi^{n-1}
\end{aligned}
\tag{2.7-3}
$$

Equation 2.7-3 shows how su and apu are defined. Note that if more than one scalar is being solved, a conditional if statement can be used in your UDF to define a different unsteady term for each i. i = 0 is associated with scalar-0 (the first scalar equation being solved).

Example

The following UDF, named my_uds_unsteady, modifies user-defined scalar time derivatives using DEFINE_UDS_UNSTEADY. The source code can be interpreted or compiled in ANSYS FLUENT.

```
/************************************************************************
   UDF for specifying user-defined scalar time derivatives
 ************************************************************************/

#include "udf.h"

DEFINE_UDS_UNSTEADY(my_uds_unsteady,c,t,i,apu,su)
{
  real physical_dt, vol, rho, phi_old;
  physical_dt = RP_Get_Real("physical-time-step");
  vol = C_VOLUME(c,t);
```

```
   rho = C_R_M1(c,t);
   *apu = -rho*vol / physical_dt;/*implicit part*/
   phi_old = C_STORAGE_R(c,t,SV_UDSI_M1(i));
   *su  = rho*vol*phi_old/physical_dt;/*explicit part*/
}
```

Hooking a UDS Unsteady Function to ANSYS FLUENT

After the UDF that you have defined using DEFINE_UDS_UNSTEADY is interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs), the name of the argument that you supplied as the first DEFINE macro argument (e.g., my_uds_unsteady) will become visible and selectable in the User-Defined Scalars dialog box in ANSYS FLUENT. See Section 6.6.3: Hooking DEFINE_UDS_UNSTEADY UDFs for details.

Chapter 3.   Additional Macros for Writing UDFs

This chapter provides predefined macros that you can use when defining your user-defined function (UDF).

3.1   Introduction

ANSYS FLUENT provides numerous C types, functions, and preprocessor macros to facilitate the programming of UDFs and the use of CFD objects as defined inside ANSYS FLUENT. The previous chapter presented DEFINE macros with which you must define your UDF. This chapter presents predefined functions (implemented as macros in the code) that are supplied by ANSYS FLUENT that you will use to code your UDF. These macros allow you to access data in an ANSYS FLUENT solver such as cell variables (e.g., cell temperature, centroid), face variables (e.g., face temperature, area), or connectivity variables (e.g., adjacent cell thread and index) that your UDF can use in a computation. A special set of macros commonly used in UDFs is provided that return such values as the thread ID pointer (an internal ANSYS FLUENT structure) when passed the Zone ID (the number assigned to a zone in a boundary conditions dialog box). Another special macro (F_PROFILE) enables your UDF to set a boundary condition value in the solver. Other types of macros are provided that enable your function to loop over nodes, cells, and faces in a thread or domain in order to retrieve and/or set values. Finally, data access macros that are specific to a particular model (e.g., DPM, NO_x) are presented, as well as macros that perform vector, time-dependent, Scheme, and I/O operations.

Function definitions for the macros provided in this chapter are contained in header files. Header files are identified by the `.h` suffix as in `mem.h`, `metric.h`, and `dpm.h` and are stored in the source code folder:

$$path\backslash\texttt{ANSYS Inc}\backslash\texttt{v120}\backslash\texttt{fluent}\backslash\texttt{fluent12.0.}\overset{\Downarrow}{x}\backslash\texttt{src}\backslash\texttt{udf.h}$$

where *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is `C:\Program Files`), and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

The header files, unless explicitly noted, are included in the `udf.h` file, so your UDF does not need to contain a special `#include` compiler directive. You must, however, remember to include the `#include "udf.h"` directive in any UDF that you write.

Access to data from an ANSYS FLUENT solver is accomplished by hooking your UDF C function (after it is compiled or interpreted) to the code through the graphical user interface (GUI). After the UDF is correctly hooked, the solver's data is passed to the function and is available to use whenever it is called. These data are automatically passed by the solver to your UDF as function arguments. Note that all solver data, regardless of whether they are passed to your UDF by the solver or returned to the solver by the UDF, are specified in SI units. Macros in this chapter are listed with their arguments, argument types, returned value(s), if applicable, and header file.

Each function behind a macro either outputs a value to the solver as an argument, or returns a value that is then available for assignment in your UDF. Input arguments belong to the following ANSYS FLUENT data types:

`Node *node`	pointer to a node
`cell_t c`	cell identifier
`face_t f`	face identifier
`Thread *t`	pointer to a thread
`Thread **pt`	pointer to an array of phase threads

Below is an example of a UDF that assigns initial temperature, which utilizes two data access macros (`C_T` and `C_CENTROID`) and two looping macros (`begin..end_c_loop_all` and `thread_loop_c`). Two looping macros are used to set the cell temperature of *each* cell in *every* thread in the computational domain. `begin..end_c_loop_all` is used to loop over all the cells in a cell thread to get the cell centroid and set the cell temperature, and `thread_loop_c` allows this loop to be repeated over all cell threads in the domain.

`C_CENTROID` has three arguments: `xc`, `c`, and `t`. Cell identifier `c` and cell thread pointer `t` are input arguments, and the argument array `xc` (the cell centroid) is output (as an argument) to the solver and used in the UDF in a conditional test.

`C_T` is used to set the cell temperature to the value of 400 or 300, depending on the outcome of the conditional test. It is passed the cell's ID `c` and thread pointer `t` and returns the `real` value of the cell temperature to the ANSYS FLUENT solver.

Example

```
/************************************************************************
   UDF for initializing flow field variables
   Example of C_T and C_CENTROID usage.
************************************************************************/

#include "udf.h"

DEFINE_INIT(my_init_func,d)
{
  cell_t c;
  Thread *t;
  real xc[ND_ND];

  /* loop over all cell threads in the domain  */
  thread_loop_c(t,d)
    {

      /* loop over all cells  */
      begin_c_loop_all(c,t)
        {
          C_CENTROID(xc,c,t);
          if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                          pow(xc[1] - 0.5,2.),
                          pow(xc[2] - 0.5,2.))) < 0.25)
            C_T(c,t) = 400.;
          else
            C_T(c,t) = 300.;
        }
      end_c_loop_all(c,t)
    }
}
```

3.2 Data Access Macros

The macros presented in this section access ANSYS FLUENT data that you can utilize in your UDF. Unless indicated, these macros can be used in UDFs for single-phase and multiphase applications.

3.2.1 Axisymmetric Considerations for Data Access Macros

C-side calculations for axisymmetric models in ANSYS FLUENT are made on a 1 radian basis. Therefore, when you are utilizing certain data access macros (e.g., `F_AREA` or `F_FLUX`) for axisymmetric flows, your UDF will need to multiply the result by 2*PI (utilizing the macro M_PI) to get the desired value.

3.2.2   Node Macros

A mesh in ANSYS FLUENT is defined by the position of its nodes and how the nodes are connected. The macros listed in Table 3.2.1 and Table 3.2.2 can be used to return the `real` Cartesian coordinates of the cell node (at the cell corner) in SI units. The variables are available in both the pressure-based and the density-based solver. Definitions for these macros can be found in `metric.h`. The argument `Node *node` for each of the variables defines a node.

Node Position

Table 3.2.1: Macros for Node Coordinates Defined in `metric.h`

Macro	Argument Types	Returns
NODE_X(node)	Node *node	real x coordinate of node
NODE_Y(node)	Node *node	real y coordinate of node
NODE_Z(node)	Node *node	real z coordinate of node

Number of Nodes in a Face (`F_NNODES`)

The macro `F_NNODES` shown in Table 3.2.2 returns the integer number of nodes associated with a face.

Table 3.2.2: Macro for Number of Nodes Defined in `mem.h`

Macro	Argument Types	Returns
F_NNODES(f,t)	face_t f, Thread *t	int number of nodes in a face

3.2.3 Cell Macros

The macros listed in Table 3.2.3–3.2.19 can be used to return `real` cell variables in SI units. They are identified by the `C_` prefix. These variables are available in the pressure-based and the density-based solver. The quantities that are returned are available only if the corresponding physical model is active. For example, species mass fraction is available only if species transport has been enabled in the Species Model dialog box in ANSYS FLUENT. Definitions for these macros can be found in the referenced header file (e.g., `mem.h`).

Cell Centroid (`C_CENTROID`)

The macro listed in Table 3.2.3 can be used to obtain the `real` centroid of a cell. `C_CENTROID` finds the coordinate position of the centroid of the cell `c` and stores the coordinates in the `x` array. Note that the `x` array is always one-dimensional, but it can be `x[2]` or `x[3]` depending on whether you are using the 2D or 3D solver.

Table 3.2.3: Macro for Cell Centroids Defined in `metric.h`

Macro	Argument Types	Outputs
`C_CENTROID(x,c,t)`	`real x[ND_ND]`, `cell_t c`, `Thread * t`	x (cell centroid)

See Section 2.2.8: `DEFINE_INIT` for an example UDF that utilizes `C_CENTROID`.

Cell Volume (`C_VOLUME`)

The macro listed in Table 3.2.4 can be used to obtain the `real` cell volume for 2D, 3D, and axisymmetric simulations.

Table 3.2.4: Macro for Cell Volume Defined in `mem.h`

Macro	Argument Types	Returns
`C_VOLUME(c,t)`	`cell_t c`, `Thread *t`	`real` cell volume for 2D or 3D, `real` cell volume/2π for axisymmetric

See Section 2.7.4: `DEFINE_UDS_UNSTEADY` for an example UDF that utilizes `C_VOLUME`.

Number of Faces (`C_NFACES`) and Nodes (`C_NNODES`) in a Cell

The macro `C_NFACES` shown in Table 3.2.5 returns the integer number of faces for a given cell. `C_NNODES`, also shown in Table 3.2.2, returns the integer number of nodes for a given cell.

Table 3.2.5: Macros for Number of Node and Faces Defined in `mem.h`

Macro	Argument Types	Returns
`C_NNODES(c,t)`	`cell_t c`, `Thread *t`	int number of nodes in a cell
`C_NFACES(c,t)`	`cell_t c`, `Thread *t`	int number of faces in a cell

Cell Face Index (`C_FACE`)

`C_FACE` expands to return the global face index `face_t f` for the given `cell_t c`, `Thread *t`, and local face index number `i`. Specific faces can be accessed via the integer index `i` and all faces can be looped over with `c_face_loop`. The macro is defined in `mem.h`.

Table 3.2.6: Macro for Cell Face Index Defined in `mem.h`

Macro	Argument Types	Returns
`C_FACE(c,t,i)`	`cell_t c`, `Thread *t`, `int i`	global face index `face_t f`

Cell Face Index (`C_FACE_THREAD`)

`C_FACE_THREAD` expands to return the `Thread *t` of the `face_t f` that is returned by `C_FACE` (see above). Specific faces can be accessed via the integer index `i` and all faces can be looped over with `c_face_loop`. The macro is defined in `mem.h`.

Table 3.2.7: Macro for Cell Face Index Defined in `mem.h`

Macro	Argument Types	Returns
`C_FACE_THREAD`	`cell_t c`, `Thread *t`, `int i`	`Thread *t` of `face_t` freturned by `C_FACE`.

Flow Variable Macros for Cells

You can access flow variables using macros listed in Table 3.2.8.

Table 3.2.8: Macros for Cell Flow Variables Defined in `mem.h`

Macro	Argument Types	Returns
`C_R(c,t)`	`cell_t c`, `Thread *t`	density
`C_P(c,t)`	`cell_t c`, `Thread *t`	pressure
`C_U(c,t)`	`cell_t c`, `Thread *t`	u velocity
`C_V(c,t)`	`cell_t c`, `Thread *t`	v velocity
`C_W(c,t)`	`cell_t c`, `Thread *t`	w velocity
`C_T(c,t)`	`cell_t c`, `Thread *t`	temperature
`C_H(c,t)`	`cell_t c`, `Thread *t`	enthalpy
`C_K(c,t)`	`cell_t c`, `Thread *t`	turb. kinetic energy
`C_NUT(c,t)`	`cell_t c`, `Thread *t`	turbulent viscosity for Spalart-Allmaras
`C_D(c,t)`	`cell_t c`, `Thread *t`	turb. kinetic energy dissipation rate
`C_O(c,t)`	`cell_t c`, `Thread *t`	specific dissipation rate
`C_YI(c,t,i)`	`cell_t c`, `Thread *t`, `int i` note: `int i` is species index	species mass fraction

Gradient (G) and Reconstruction Gradient (RG) Vector Macros

You can access gradient and reconstruction gradient vectors (and components) for many of the cell variables listed in Table 3.2.8. ANSYS FLUENT calculates the gradient of flow in a cell (based on the divergence theory) and stores this value in the variable identified by the suffix `_G`. For example cell temperature is stored in the variable `C_T`, and the temperature gradient of the cell is stored in `C_T_G`. The gradients stored in variables with the `_G` suffix are non-limited values and if used to reconstruct values within the cell (at faces, for example), may potentially result in values that are higher (or lower) than values in the surrounding cells. Therefore, if your UDF needs to compute face values from cell gradients, you should use the reconstruction gradient (RG) values instead of non-limited gradient (G) values. Reconstruction gradient variables are identified by the suffix `_RG`, and use the limiting method that you have activated in your ANSYS FLUENT model to limit the cell gradient values.

Gradient (G) Vector Macros

Table 3.2.9 shows a list of cell gradient vector macros. Note that gradient variables are available *only* when the equation for that variable is being solved. For example, if you are defining a source term for energy, your UDF can access the cell temperature gradient (using C_T_G), but it cannot get access to the x-velocity gradient (using C_U_G). The reason for this is that the solver continually removes data from memory that it doesn't need. In order to retain the gradient data (when you want to set up user-defined scalar transport equations, for example), you can prevent the solver from freeing up memory by issuing the text command `solve/set/expert` and then answering `yes` to the question `Keep temporary solver memory from being freed?`. Note that when you do this, all of the gradient data is retained, but the calculation requires more memory to run.

You can access a component of a gradient vector by specifying it as an argument in the gradient vector call (0 for the x component; 1 for y; and 2 for z). For example,

```
C_T_G(c,t)[0];    /* returns the x-component of the cell temperature
                      gradient vector */
```

returns the x component of the temperature gradient vector.

Table 3.2.9: Macros for Cell Gradients Defined in `mem.h`

Macro	Argument Types	Returns
C_R_G(c,t)	cell_t c, Thread *t	density gradient vector
C_P_G(c,t)	cell_t c, Thread *t	pressure gradient vector
C_U_G(c,t)	cell_t c, Thread *t	velocity gradient vector
C_V_G(c,t)	cell_t c, Thread *t	velocity gradient vector
C_W_G(c,t)	cell_t c, Thread *t	velocity gradient vector
C_T_G(c,t)	cell_t c, Thread *t	temperature gradient vector
C_H_G(c,t)	cell_t c, Thread *t	enthalpy gradient vector
C_NUT_G(c,t)	cell_t c, Thread *t	turbulent viscosity for Spalart-Allmaras gradient vector
C_K_G(c,t)	cell_t c, Thread *t	turbulent kinetic energy gradient vector
C_D_G(c,t)	cell_t c, Thread *t	turbulent kinetic energy dissipation rate gradient vector
C_O_G(c,t)	cell_t c, Thread *t	specific dissipation rate gradient vector
C_YI_G(c,t,i)	cell_t c, Thread *t, int i note: int i is species index	species mass fraction gradient vector

i   Note that you can access vector components of each of the variables listed in Table 3.2.9 by using the integer index [i] for each macro listed in Table 3.2.9. For example, C_T_G(c,t)[i] will access a component of the temperature gradient vector.

i   C_R_G can be used only in the density-based solver, and C_P_G can be used only in the pressure-based solver.

i   C_YI_G can be used only in the density-based solver. To use this in the pressure-based solver, you will need to set the rpvar 'species/save-gradients? to #t.

As stated previously, the availability of gradient variables is affected by your solver selection, which models are turned on, the setting for the spatial discretization, and whether the temporary solver memory is retained. To make it easy for you to verify what gradient variables are available for your particular case and data files, the following UDF (named showgrad.c) is provided. Simply compile this UDF, run your solution, and then hook the UDF using the **Execute on Demand** dialog box (as described in Section 6.1.6: Hooking DEFINE_ON_DEMAND UDFs). The available gradient variables will be displayed in the console.

i   Note that the showgrad.c UDF is only useful for single phase models.

```
/*
 *  ON Demand User-Defined Functions to check                       *
 *  on the availability of Reconstruction Gradient and Gradients    *
 *  for a given Solver and Solver settings:                         *
 *                                                                  *
 *  Availability of Gradients & Reconstruction Gradients depends on: *
 *  1) the selected Solver (density based or pressure based)        *
 *  2) the selected Model                                           *
 *  3) the order of discretizations                                 *
 *  4) whether the temporary solver memory is being retained (to keep *
 *     temporary memory go to solve -> set ->expert and type YES    *
 *     for "Keep temporary solver memory from being freed?")        *
 *                                                                  *
 *                                                                  *
 *  How to use showgrad:                                            *
 *                                                                  *
 *  - Read in your case & data file.                                *
 *  - Compile showgrad.c UDF.                                       *
 *  - Load library libudf.                                          *
```

```
*  - Attach the showgrad UDF in the Execute on Demand dialog box.      *
*  - Run your solution.                                                *
*  - Click the Execute button in the Execute on Demand dialog box.     *
*                                                                      *
*  A list of available Grads and Recon Grads will be displayed in the  *
*  console.                                                            *
*                                                                      *
*  2004 Laith Zori                                                     *
*/
#include "udf.h"

DEFINE_ON_DEMAND(showgrad)
{
  Domain *domain;
  Thread *t;
  domain=Get_Domain(1);

  if (! Data_Valid_P())     return;

  Message0(">>> entering show-grad:  \n ");

  thread_loop_c(t, domain)
    {
      Material *m = THREAD_MATERIAL(t);
      int nspe    = MIXTURE_NSPECIES(m);
      int nspm    = nspe-1;

      Message0("::::\n ");
      Message0("::::   Reconstruction Gradients    :::: \n ");
      Message0("::::\n ");

      if (NNULLP(THREAD_STORAGE(t, SV_P_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of P is available \n ");
}

      if (NNULLP(THREAD_STORAGE(t, SV_U_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of U is available \n ");
}

      if (NNULLP(THREAD_STORAGE(t, SV_V_RG  )))
{
```

```
    Message0("....show-grad:Reconstruction Gradient of V is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_W_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of W is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_T_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of T is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_H_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of H is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_K_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of K is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_D_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of D is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_O_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of O is available \n ");
}


    if (NNULLP(THREAD_STORAGE(t, SV_NUT_RG  )))
{
  Message0("....show-grad:Reconstruction Gradient of NUT is available \n ");
}


    if (nspe)
    {
int ns     = 0     ;
spe_loop (ns,nspm)
```

```
      if (NNULLP(THREAD_STORAGE(t, SV_Y_I(ns)+SV_Y_0_RG-SV_Y_0  )))
        {
          Message0("....show-grad:Reconstruction Gradient of Species %d is "
                       "available \n ",ns);
        }
          }


      /*********************************************************************/
      /*********************************************************************/
      /*********************************************************************/
      /*********************************************************************/

      Message0("::::\n ");
      Message0("::::         Gradients     :::: \n ");
      Message0("::::\n ");

      if (NNULLP(THREAD_STORAGE(t, SV_P_G  )))
{
  Message0("....show-grad:Gradient of P is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_U_G  )))
{
  Message0("....show-grad:Gradient of U is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_V_G  )))
{
  Message0("....show-grad:Gradient of V is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_W_G  )))
{
  Message0("....show-grad:Gradient of W is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_T_G  )))
{
  Message0("....show-grad:Gradient of T is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_H_G  )))
```

```
{
  Message0("....show-grad:Gradient of H is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_K_G  )))
{
  Message0("....show-grad:Gradient of K is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_D_G  )))
{
  Message0("....show-grad:Gradient of D is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_O_G  )))
{
  Message0("....show-grad:Gradient of O is available \n ");
}


      if (NNULLP(THREAD_STORAGE(t, SV_NUT_G  )))
{
  Message0("....show-grad:Gradient of NUT is available \n ");
}



      if (nspe)
{
  int ns      = 0      ;
  spe_loop (ns,nspm)
    if (NNULLP(THREAD_STORAGE(t, SV_Y_I(ns)+SV_Y_0_G-SV_Y_0  )))
      {
Message0("....show-grad:Gradient of Species %d is available \n ",ns);
      }
}


    }
}
```

Reconstruction Gradient (RG) Vector Macros

Table 3.2.10 shows a list of cell reconstruction gradient vector macros. Like gradient variables, RG variables are available only when the equation for that variable is being solved. As in the case of gradient variables, you can retain all of the reconstruction gradient data by issuing the text command `solve/set/expert` and then answering `yes` to the question `Keep temporary solver memory from being freed?`. Note that when you do this, the reconstruction gradient data is retained, but the calculation requires more memory to run.

You can access a component of a reconstruction gradient vector by specifying it as an argument in the reconstruction gradient vector call (`0` for the x component; `1` for y; and `2` for z). For example,

```
C_T_RG(c,t)[0];    /* returns the x-component of the cell temperature
                       reconstruction gradient vector */
```

returns the x component of the temperature reconstruction gradient vector.

Table 3.2.10: Macros for Cell Reconstruction Gradients (RG) Defined in `mem.h`

Macro	Argument Types	Returns
C_R_RG(c,t)	cell_t c, Thread *t	density RG vector
C_P_RG(c,t)	cell_t c, Thread *t	pressure RG vector
C_U_RG(c,t)	cell_t c, Thread *t	velocity RG vector
C_V_RG(c,t)	cell_t c, Thread *t	velocity RG vector
C_W_RG(c,t)	cell_t c, Thread *t	velocity RG vector
C_T_RG(c,t)	cell_t c, Thread *t	temperature RG vector
C_H_RG(c,t)	cell_t c, Thread *t	enthalpy RG vector
C_NUT_RG(c,t)	cell_t c, Thread *t	turbulent viscosity for Spalart-Allmaras RG vector
C_K_RG(c,t)	cell_t c, Thread *t	turbulent kinetic energy RG vector
C_D_RG(c,t)	cell_t c, Thread *t	turbulent kinetic energy dissipation rate RG vector
C_YI_RG(c,t,i)	cell_t c, Thread *t, int i note: int i is species index	species mass fraction RG vector

i Note that you can access vector components by using the integer index `[i]` for each macro listed in Table 3.2.10. For example, `C_T_RG(c,t)[i]` will access a component of the temperature reconstruction gradient vector.

> *i*   C_P_RG can be used in the pressure-based solver only when the second order discretization scheme for pressure is specified.

> *i*   C_YI_RG can be used only in the density-based solver.

As stated previously, the availability of reconstruction gradient variables is affected by your solver selection, which models are turned on, the setting for the spatial discretization, and whether the temporary solver memory is freed. To make it easy for you to verify which reconstruction gradient variables are available for your particular case and data files, a UDF (named `showgrad.c`) has been provided that will display the available gradients in the console. See the previous section for details.

Previous Time Step Macros

The _M1 suffix can be applied to some of the cell variable macros in Table 3.2.8 to allow access to the value of the variable at the previous time step (i.e., $t - \Delta t$). These data may be useful in unsteady simulations. For example,

```
C_T_M1(c,t);
```

returns the value of the cell temperature at the previous time step. Previous time step macros are shown in Table 3.2.11.

> *i*   Note that data from C_T_M1 is available *only* if user-defined scalars are defined. It can also be used with adaptive time stepping.

Table 3.2.11: Macros for Cell Time Level 1 Defined in `mem.h`

Macro	Argument Types	Returns
C_R_M1(c,t)	cell_t c, Thread *t	density, previous time step
C_P_M1(c,t)	cell_t c, Thread *t	pressure, previous time step
C_U_M1(c,t)	cell_t c, Thread *t	velocity, previous time step
C_V_M1(c,t)	cell_t c, Thread *t	velocity, previous time step
C_W_M1(c,t)	cell_t c, Thread *t	velocity, previous time step
C_T_M1(c,t)	cell_t c, Thread *t	temperature, previous time step
C_YI_M1(c,t,i)	cell_t c, Thread *t, int i note: int i is species index	species mass fraction, previous time step

See Section 2.7.4: DEFINE_UDS_UNSTEADY for an example UDF that utilizes C_R_M1.

The `M2` suffix can be applied to some of the cell variable macros in Table 3.2.11 to allow access to the value of the variable at the time step before the previous one (i.e., $t - 2\Delta t$). These data may be useful in unsteady simulations. For example,

```
C_T_M2(c,t);
```

returns the value of the cell temperature at the time step before the previous one (referred to as second previous time step). Two previous time step macros are shown in Table 3.2.12.

i   Note that data from `C_T_M2` is available *only* if user-defined scalars are defined. It can also be used with adaptive time stepping.

Table 3.2.12: Macros for Cell Time Level 2 Defined in `mem.h`

Macro	Argument Types	Returns
`C_R_M2(c,t)`	`cell_t c, Thread *t`	density, second previous time step
`C_P_M2(c,t)`	`cell_t c, Thread *t`	pressure, second previous time step
`C_U_M2(c,t)`	`cell_t c, Thread *t`	velocity, second previous time step
`C_V_M2(c,t)`	`cell_t c, Thread *t`	velocity, second previous time step
`C_W_M2(c,t)`	`cell_t c, Thread *t`	velocity, second previous time step
`C_T_M2(c,t)`	`cell_t c, Thread *t`	temperature, second previous time step
`C_YI_M2(c,t,i)`	`cell_t c, Thread *t, int i`	species mass fraction, second previous time step

Derivative Macros

The macros listed in Table 3.2.13 can be used to return `real` velocity derivative variables in SI units. The variables are available in both the pressure-based and the density-based solver. Definitions for these macros can be found in the `mem.h` header file.

Table 3.2.13: Macros for Cell Velocity Derivatives Defined in `mem.h`

Macro	Argument Types	Returns
C_STRAIN_RATE_MAG(c,t)	cell_t c, Thread *t	strain rate magnitude
C_DUDX(c,t)	cell_t c, Thread *t	velocity derivative
C_DUDY(c,t)	cell_t c, Thread *t	velocity derivative
C_DUDZ(c,t)	cell_t c, Thread *t	velocity derivative
C_DVDX(c,t)	cell_t c, Thread *t	velocity derivative
C_DVDY(c,t)	cell_t c, Thread *t	velocity derivative
C_DVDZ(c,t)	cell_t c, Thread *t	velocity derivative
C_DWDX(c,t)	cell_t c, Thread *t	velocity derivative
C_DWDY(c,t)	cell_t c, Thread *t	velocity derivative
C_DWDZ(c,t)	cell_t c, Thread *t	velocity derivative

Material Property Macros

The macros listed in Tables 3.2.14–3.2.16 can be used to return `real` material property variables in SI units. The variables are available in both the pressure-based and the density-based solver. Argument `real prt` is the turbulent Prandtl number. Definitions for material property macros can be found in the referenced header file (e.g., `mem.h`).

Table 3.2.14: Macros for Diffusion Coefficients Defined in `mem.h`

Macro	Argument Types	Returns
`C_MU_L(c,t)`	`cell_t c, Thread *t`	laminar viscosity
`C_MU_T(c,t)`	`cell_t c, Thread *t`	turbulent viscosity
`C_MU_EFF(c,t)`	`cell_t c, Thread *t`	effective viscosity
`C_K_L(c,t)`	`cell_t c, Thread *t`	thermal conductivity
`C_K_T(c,t,prt)`	`cell_t c, Thread *t, real prt`	turbulent thermal conductivity
`C_K_EFF(c,t,prt)`	`cell_t c, Thread *t, real prt`	effective thermal conductivity
`C_DIFF_L(c,t,i,j)`	`cell_t c, Thread *t, int i, int j`	laminar species diffusivity
`C_DIFF_EFF(c,t,i)`	`cell_t c, Thread *t, int i`	effective species diffusivity

Table 3.2.15: Macros for Thermodynamic Properties Defined in `mem.h`

Name(Arguments)	Argument Types	Returns
`C_CP(c,t)`	`cell_t c, Thread *t`	specific heat
`C_RGAS(c,t)`	`cell_t c, Thread *t`	universal gas constant/molecular weight
`C_NUT(c,t)`	`cell_t c, Thread *t`	turbulent viscosity for Spalart-Allmaras

Table 3.2.16: Additional Material Property Macros Defined in `sg_mem.h`

Macro	Argument Types	Returns
C_FMEAN(c,t)	cell_t c, Thread *t	primary mean mixture fraction
C_FMEAN2(c,t)	cell_t c, Thread *t	secondary mean mixture fraction
C_FVAR(c,t)	cell_t c, Thread *t	primary mixture fraction variance
C_FVAR2(c,t)	cell_t c, Thread *t	secondary mixture fraction variance
C_PREMIXC(c,t)	cell_t c, Thread *t	reaction progress variable
C_LAM_FLAME_SPEED(c,t)	cell_t c, Thread *t	laminar flame speed
C_SCAT_COEFF(c,t)	cell_t c, Thread *t	scattering coefficient
C_ABS_COEFF(c,t)	cell_t c, Thread *t	absorption coefficient
C_CRITICAL_STRAIN_RATE(c,t)	cell_t c, Thread *t	critical strain rate
C_LIQF(c,t)	cell_t c, Thread *t	liquid fraction in a cell
C_POLLUT(c,t,i)	cell_t c, Thread *t, int i	ith pollutant species mass fraction (see table below)

i   C_LIQF is available only in fluid cells and only if solidification is turned ON.

Table 3.2.17: Table of Definitions for Argument `i` of the Pollutant Species
Mass Fraction Function `C_POLLUT`

i	Definitions
0	Mass Fraction of NO
1	Mass Fraction of HCN
2	Mass Fraction of NH3
3	Mass Fraction of N2O
4	Soot Mass Fraction
5	Normalized Radical Nuclei

Note: Concentration in particles $\times 10^{-15}$/kg. For mass fraction concentrations in the table above, see Equation 13.3-7 in the separate Theory Guide for the defining equation.

Reynolds Stress Model Macros

The macros listed in Table 3.2.18 can be used to return `real` variables for the Reynolds stress turbulence model in SI units. The variables are available in both the pressure-based and the density-based solver. Definitions for these macros can be found in the `metric.h` header file.

Table 3.2.18: Macros for Reynolds Stress Model Variables Defined in `sg_mem.h`

Macro	Argument Types	Returns
`C_RUU(c,t)`	`cell_t c`, `Thread *t`	uu Reynolds stress
`C_RVV(c,t)`	`cell_t c`, `Thread *t`	vv Reynolds stress
`C_RWW(c,t)`	`cell_t c`, `Thread *t`	ww Reynolds stress
`C_RUV(c,t)`	`cell_t c`, `Thread *t`	uv Reynolds stress
`C_RVW(c,t)`	`cell_t c`, `Thread *t`	vw Reynolds stress
`C_RUW(c,t)`	`cell_t c`, `Thread *t`	uw Reynolds stress

VOF Multiphase Model Macro

The macro `C_VOF` can be used to return `real` variables associated with the VOF multiphase model in SI units. The variables are available in both the pressure-based and the density-based solver, with the exception of the VOF variable, which is available only for the pressure-based solver. Definitions for these macros can be found in `sg_mphase.h`, which is included in `udf.h`.

Table 3.2.19: Macros for Multiphase Variables Defined in `sg_mphase.h`

Macro	Argument Types	Returns
C_VOF(c,t)	cell_t c, Thread *t (has to be a phase thread)	volume fraction for the phase corresponding to phase thread t.

3.2.4 Face Macros

The macros listed in Table 3.2.20–3.2.23 can be used to return `real` face variables in SI units. They are identified by the `F_` prefix. Note that these variables are available *only* in the pressure-based solver. In addition, quantities that are returned are available only if the corresponding physical model is active. For example, species mass fraction is available only if species transport has been enabled in the Species Model dialog box in ANSYS FLUENT. Definitions for these macros can be found in the referenced header files (e.g., `mem.h`).

Face Centroid (`F_CENTROID`)

The macro listed in Table 3.2.20 can be used to obtain the `real` centroid of a face. `F_CENTROID` finds the coordinate position of the centroid of the face `f` and stores the coordinates in the `x` array. Note that the `x` array is always one-dimensional, but it can be `x[2]` or `x[3]` depending on whether you are using the 2D or 3D solver.

Table 3.2.20: Macro for Face Centroids Defined in `metric.h`

Macro	Argument Types	Outputs
F_CENTROID(x,f,t)	real x[ND_ND], face_t f, Thread *t	x (face centroid)

The ND_ND macro returns 2 or 3 in 2D and 3D cases, respectively, as defined in Section 3.4.2: The ND Macros. Section 2.3.15: DEFINE_PROFILE contains an example of F_CENTROID usage.

Face Area Vector (F_AREA)

F_AREA can be used to return the `real` face area vector (or 'face area normal') of a given face f in a face thread t. See Section 2.7.3: DEFINE_UDS_FLUX for an example UDF that utilizes F_AREA.

Table 3.2.21: Macro for Face Area Vector Defined in `metric.h`

Macro	Argument Types	Outputs
F_AREA(A,f,t)	A[ND_ND], face_t f, Thread *t	A (area vector)

By convention in ANSYS FLUENT, boundary face area normals always point out of the domain. ANSYS FLUENT determines the direction of the face area normals for interior faces by applying the right hand rule to the nodes on a face, in order of increasing node number. This is shown in Figure 3.2.1.
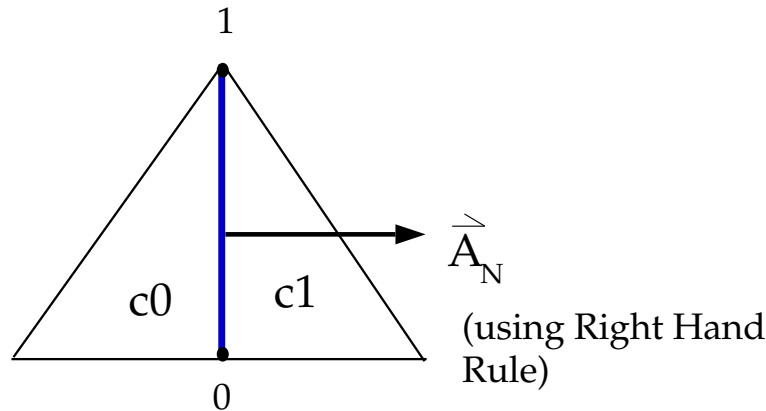
Figure 3.2.1: ANSYS FLUENT Determination of Face Area Normal Direction: 2D Face

ANSYS FLUENT assigns adjacent cells to an interior face (c0 and c1) according to the following convention: the cell *out* of which a face area normal is pointing is designated as cell C0, while the cell *in* to which a face area normal is pointing is cell c1 (Figure 3.2.1). In other words, face area normals always point from cell c0 to cell c1.

Flow Variable Macros for Boundary Faces

The macros listed in Table 3.2.22 access flow variables at a boundary face.

Table 3.2.22: Macros for Boundary Face Flow Variables Defined in `mem.h`

Macro	Argument Types	Returns
`F_U(f,t)`	`face_t f, Thread *t,`	u velocity
`F_V(f,t)`	`face_t f, Thread *t,`	v velocity
`F_W(f,t)`	`face_t f, Thread *t,`	w velocity
`F_T(f,t)`	`face_t f, Thread *t,`	temperature
`F_H(f,t)`	`face_t f, Thread *t,`	enthalpy
`F_K(f t)`	`face_t f, Thread *t,`	turbulent kinetic energy
`F_D(f,t)`	`face_t f, Thread *t,`	turbulent kinetic energy dissipation rate
`F_YI(f,t,i)`	`face_t f, Thread *t, int i`	species mass fraction

See Section 2.7.3: `DEFINE_UDS_FLUX` for an example UDF that utilizes some of these macros.

Flow Variable Macros at Interior and Boundary Faces

The macros listed in Table 3.2.23 access flow variables at interior faces and boundary faces.

Table 3.2.23: Macros for Interior and Boundary Face Flow Variables Defined in `mem.h`

Macro	Argument Types	Returns
`F_P(f,t)`	`face_t f, Thread *t,`	pressure
`F_FLUX(f,t)`	`face_t f, Thread *t`	mass flow rate through a face

`F_FLUX` can be used to return the `real` scalar mass flow rate through a given face `f` in a face thread `t`. The sign of `F_FLUX` that is computed by the ANSYS FLUENT solver is positive if the flow direction is the same as the face area normal direction (as determined by `F_AREA` - see Section 3.2.4: Face Area Vector (`F_AREA`)), and is negative if the flow

direction and the face area normal directions are opposite. In other words, the flux is positive if the flow is *out* of the domain, and is negative if the flow is *in* to the domain.

Note that the sign of the flux that is computed by the solver is opposite to that which is reported in the ANSYS FLUENT GUI (e.g., the Flux Reports dialog box).

3.2.5 Connectivity Macros

ANSYS FLUENT provides macros that allow the vectors connecting cell centroids and the vectors connecting cell and face centroids to be readily defined. These macros return information that is helpful in evaluating face values of scalars which are generally not stored, as well as the diffusive flux of scalars across cell boundaries. The geometry and gradients involved with these macros are summarized in Figure 3.2.2.

To better understand the parameters that are returned by these macros, it is best to consider how the aforementioned calculations are evaluated. Assuming that the gradient of a scalar is available, the face value of a scalar, ϕ, can be approximated by

$$\phi_f = \phi_0 + \nabla\phi \cdot \vec{dr} \tag{3.2-1}$$

where $\vec{dr}$ is the vector that connects the cell centroid with the face centroid. The gradient in this case is evaluated at the cell centroid where ϕ_0 is also stored.

The diffusive flux, D_f, across a face, f, of a scalar ϕ is given by,

$$D_f = \Gamma_f \nabla\phi \cdot \vec{A} \tag{3.2-2}$$

where Γ_f is the diffusion coefficient at the face. In ANSYS FLUENT's unstructured solver, the gradient along the face normal direction may be approximated by evaluating gradients along the directions that connect cell centroids and along a direction confined within the plane of the face. Given this, D_f may be approximated as,

$$D_f = \Gamma_f \frac{(\phi_1 - \phi_0)}{ds} \frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e_s}} + \Gamma_f \left(\overline{\nabla}\phi \cdot \vec{A} - \overline{\nabla}\phi \cdot \vec{e_s} \frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e_s}} \right) \tag{3.2-3}$$

where the first term on the right hand side represents the primary gradient directed along the vector $\vec{e_s}$ and the second term represents the 'cross' diffusion term. In this equation, A is the area normal vector of face f directed from cell c0 to c1, ds is the distance between the cell centroids, and $\vec{e_s}$ is the unit normal vector in this direction. $\overline{\nabla}\phi$ is the average of the gradients at the two adjacent cells. (For boundary faces, the variable is the gradient of the c0 cell.) This is shown in Figure 3.2.2.
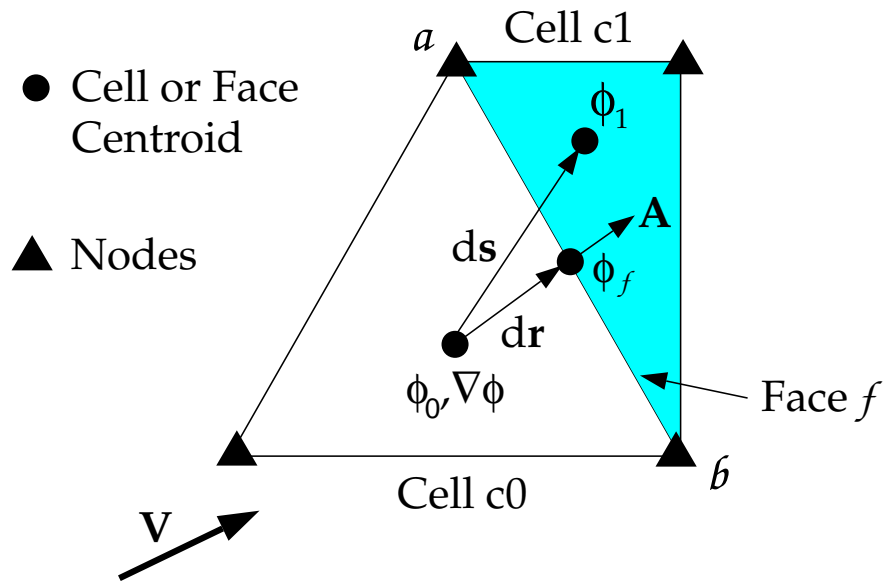
Figure 3.2.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions

Adjacent Cell Index (F_C0, F_C1)

The cells on either side of a face may or may not belong to the same cell thread. Referring to Figure 3.2.2, if a face is on the boundary of a domain, then only `c0` exists. (`c1` is undefined for an external face). Alternatively, if the face is in the interior of the domain, then both `c0` and `c1` exist.

There are two macros, `F_C0(f,t)` and `F_C1(f,t)`, that can be used to identify cells that are adjacent to a given face thread `t`. `F_C0` expands to a function that returns the index of a face's neighboring `c0` cell (Figure 3.2.2), while `F_C1` returns the cell index for `c1` (Figure 3.2.2), if it exists.

Table 3.2.24: Adjacent Cell Index Macros Defined in `mem.h`

Macro	Argument Types	Returns
F_C0(f,t)	face_t f, Thread *t	cell_t c for cell c0
F_C1(f,t)	face_t f, Thread *t	cell_t c for cell c1

See Section 2.7.3: `DEFINE_UDS_FLUX` for an example UDF that utilizes `F_C0`.

Adjacent Cell Thread (`THREAD_T0`, `THREAD_T1`)

The cells on either side of a face may or may not belong to the same cell thread. Referring to Figure 3.2.2, if a face is on the boundary of a domain, then only `c0` exists. (`c1` is undefined for an external face). Alternatively, if the face is in the interior of the domain, then both `c0` and `c1` exist.

There are two macros, `THREAD_T0(t)` and `THREAD_T1(t)`, that can be used to identify cell threads that are adjacent to a given face `f` in a face thread `t`. `THREAD_T0` expands to a function that returns the cell thread of a given face's adjacent cell `c0`, and `THREAD_T1` returns the cell thread for `c1` (if it exists).

Table 3.2.25: Adjacent Cell Thread Macros Defined in `mem.h`

Macro	Argument Types	Returns
`THREAD_T0(t)`	`Thread *t`	cell thread pointer for cell c0
`THREAD_T1(t)`	`Thread *t`	cell thread pointer for cell c1

Interior Face Geometry (`INTERIOR_FACE_GEOMETRY`)

`INTERIOR_FACE_GEOMETRY(f,t,A,ds,es,A_by_es,dr0,dr1)` expands to a function that outputs the following variables to the solver, for a given face `f`, on face thread `t`. The macro is defined in the `sg.h` header file which is *not* included in `udf.h`. You will need to include this file in your UDF using the `#include` directive.

`real A[ND_ND]`	the area normal vector
`real ds`	distance between the cell centroids
`real es[ND_ND]`	the unit normal vector in the direction from cell c0 to c1
`real A_by_es`	the value $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e_s}}$
`real dr0[ND_ND]`	vector that connects the centroid of cell `c0` to the face centroid
`real dr1[ND_ND]`	the vector that connects the centroid of cell `c1` to the face centroid

Note that `INTERIOR_FACE_GEOMETRY` can be called to retrieve some of the terms needed to evaluate Equations 3.2-1 and 3.2-3.

Boundary Face Geometry (`BOUNDARY_FACE_GEOMETRY`)

`BOUNDARY_FACE_GEOMETRY(f,t,A,ds,es,A_by_es,dr0)` expands to a function that outputs the following variables to the solver, for a given face `f`, on face thread `t`. It is defined in the `sg.h` header file which is *not* included in `udf.h`. You will need to include this file in your UDF using the `#include` directive.

BOUNDARY_FACE_GEOMETRY can be called to retrieve some of the terms needed to evaluate Equations 3.2-1 and 3.2-3.

real A[ND_ND]	area normal vector
real ds	distance between the cell centroid and the face centroid
real es[ND_ND]	unit normal vector in the direction from centroid of cell c0 to the face centroid
real A_by_es	value $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e_s}}$
real dr0[ND_ND]	vector that connects the centroid of cell c0 to the face centroid

Boundary Face Thread (BOUNDARY_FACE_THREAD)

BOUNDARY_FACE_THREAD_P(t) expands to a function that returns TRUE if Thread *t is a boundary face thread. The macro is defined in threads.h which is included in udf.h. See Section 2.7.3: DEFINE_UDS_FLUX for an example UDF that utilizes BOUNDARY_FACE_THREAD_P.

Boundary Secondary Gradient Source (BOUNDARY_SECONDARY_GRADIENT_SOURCE)

BOUNDARY_SECONDARY_GRADIENT_SOURCE(source,n,dphi,dx,A_by_es,k) expands to a function that outputs the following variables to the solver, for a given face and face thread. It is defined in the sg.h header file which is *not* included in udf.h. You will need to include this file in your UDF using the #include directive.

> **i** The use of BOUNDARY_SECONDARY_GRADIENT_SOURCE first requires that cell geometry information be defined, which can be readily obtained by the use of the BOUNDARY_FACE_GEOMETRY macro (described previously in this section). See Section 8.2.5: Implementing ANSYS FLUENT's P-1 Radiation Model Using User-Defined Scalars for an example.

BOUNDARY_SECONDARY_GRADIENT_SOURCE can be called to retrieve some of the terms needed to evaluate Equation 3.2-3.

real source	the cross diffusion term of the diffusive flux (i.e., the second term on the right side of Equation 3.2-3)
real n	the average of the facial gradient ($\overline{\nabla}\phi$ in Equation 3.2-3)
real dphi[ND_ND]	a dummy scratch variable array that stores the facial gradient value during the computation
real dx[ND_ND]	the unit normal vector in the direction from centroid of cell c0 to the face centroid
real A_by_es	the value $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e_s}}$
real k	the diffusion coefficient at the face (Γ_f in Equation 3.2-3)

$\boxed{i}$ Note that the average of the facial gradient (supplied for `n`) is not always allocated, and so your UDF must verify its status (using the `NULLP` or `NNULLP` function, as described in Section 3.8: NULLP & NNULLP) and assign a value as necessary. See Section 8.2.5: Implementing ANSYS FLUENT's P-1 Radiation Model Using User-Defined Scalars for an example.

3.2.6   Special Macros

The macros listed in this section are special macros that are used often in UDFs.

- `Lookup_Thread`

- `THREAD_ID`

- `Get_Domain`

- `F_PROFILE`

- `THREAD_SHADOW`

Thread Pointer for Zone ID (`Lookup_Thread`)

You can use `Lookup_Thread` when you want to retrieve the pointer `t` to the thread that is associated with a given integer zone ID number for a boundary zone. The `zone_ID` that is passed to the macro is the zone number that ANSYS FLUENT assigns to the boundary and displays in the boundary condition dialog box (e.g., `Fluid`). Note that this macro does the inverse of `THREAD_ID` (see below).

There are two arguments to `Lookup_Thread`. `domain` is passed by ANSYS FLUENT and is the pointer to the domain structure. You supply the integer value of `zone_ID`.

For example, the code

```
int zone_ID = 2;
Thread *thread_name = Lookup_Thread(domain,zone_ID);
```

passes a zone ID of 2 to `Lookup_Thread`. A zone ID of 2 may, for example, correspond to a wall zone in your case.

Now suppose that your UDF needs to operate on a particular thread in a domain (instead of looping over all threads), and the `DEFINE` macro you are using to define your UDF doesn't have the `thread` pointer passed to it from the solver (e.g., `DEFINE_ADJUST`). You can use `Lookup_Thread` in your UDF to get the desired thread pointer. This is a two-step process.

First, you will need to get the integer ID of the zone by visiting the boundary condition dialog box (e.g., **Fluid**) and noting the zone ID. You can also obtain the value of the Zone ID from the solver using `RP_Get_Integer`. Note that in order to use `RP_Get_Integer`, you will have had to define the zone ID variable first, either in another UDF using `RP_Set_Integer`, or on the Scheme side using `rp-var-define` (see Section 3.6: Scheme Macros for details.)

Next, you supply the `zone_ID` as an argument to `Lookup_Thread` either as a hard-coded integer (e.g., `1`, `2`) or as the variable assigned from `RP_Get_Integer`. `Lookup_Thread` returns the pointer to the `thread` that is associated with the given zone ID. You can then assign the thread pointer to a `thread_name` and use it in your UDF.

> ℹ️ Note that when `Lookup_Thread` is utilized in a multiphase flow problem, the domain pointer that is passed to the function depends on the UDF that it is contained within. For example, if `Lookup_Thread` is used in an adjust function (`DEFINE_ADJUST`) then the mixture domain is passed and the thread pointer returned is the mixture-level thread.

Example

Below is a UDF that uses `Lookup_Thread`. In this example, the pointer to the thread for a given `zone_ID` is retrieved by `Lookup_Thread` and is assigned to `thread`. The `thread` pointer is then used in `begin_f_loop` to loop over all faces in the given thread, and in `F_CENTROID` to get the face centroid value.

```
/*********************************************************************/
    Example of an adjust UDF that uses Lookup_Thread.
    Note that if this UDF is applied to a multiphase flow problem,
    the thread that is returned is the mixture-level thread
**********************************************************************/

#include "udf.h"

/* domain passed to Adjust function is mixture domain for multiphase*/

DEFINE_ADJUST(print_f_centroids, domain)
{
  real FC[2];
  face_t f;
  int ID = 1;
      /* Zone ID for wall-1 zone from Boundary Conditions task page */
  Thread *thread = Lookup_Thread(domain, ID);
  begin_f_loop(f, thread)
    {
       F_CENTROID(FC,f,thread);
```

```
      printf("x-coord = %f  y-coord = %f", FC[0], FC[1]);
    }
  end_f_loop(f,thread)
}
```

Zone ID (`THREAD_ID`)

You can use `THREAD_ID` when you want to retrieve the integer zone ID number (displayed in a boundary conditions dialog box such as `Fluid`) that is associated with a given thread pointer `t`. Note that this macro does the inverse of `Lookup_Thread` (see above).

```
int zone_ID = THREAD_ID(t);
```

Domain Pointer (`Get_Domain`)

You can use the `Get_Domain` macro to retrieve a domain pointer when it is not explicitly passed as an argument to your UDF. This is commonly used in `ON_DEMAND` functions since `DEFINE_ON_DEMAND` is not passed any arguments from the ANSYS FLUENT solver. It is also used in initialization and adjust functions for multiphase applications where a phase domain pointer is needed but only a mixture pointer is passed.

```
Get_Domain(domain_id);
```

`domain_id` is an `integer` whose value is 1 for the mixture domain, but the values for the phase domains can be any integer greater than 1. The `ID` for a particular phase can be found be selecting it in the Phases task page in ANSYS FLUENT.

✧ Phases

Single-Phase Flows

In the case of single-phase flows, `domain_id` is 1 and `Get_Domain(1)` will return the fluid domain pointer.

```
DEFINE_ON_DEMAND(my_udf)
{
  Domain *domain;         /* domain is declared as a variable   */
  domain = Get_Domain(1); /* returns fluid domain pointer       */
  ...
}
```

Multiphase Flows

In the case of multiphase flows, the value returned by `Get_Domain` is either the mixture-level, a phase-level, or an interaction phase-level domain pointer. The value of `domain_id` is always 1 for the mixture domain. You can obtain the `domain_id` using the ANSYS FLUENT graphical user interface much in the same way that you can determine the zone ID from the Boundary Conditions task page. Simply go to the Phases task page in ANSYS FLUENT and select the desired phase. The `domain_id` will then be displayed. You will need to hard code this integer ID as an argument to the macro as shown below.

```
DEFINE_ON_DEMAND(my_udf)
{
  Domain *mixture_domain;
  mixture_domain = Get_Domain(1);   /* returns mixture domain pointer */
                                    /* and assigns to variable        */

  Domain *subdomain;
  subdomain = Get_Domain(2); /* returns phase with ID=2 domain pointer*/
                             /* and assigns to variable               */
  ...
}
```

Example

The following example is a UDF named `get_coords` that prints the thread face centroids for two specified thread IDs. The function implements the `Get_Domain` utility for a single-phase application. In this example, the function `Print_Thread_Face_Centroids` uses the `Lookup_Thread` function to determine the pointer to a thread, and then writes the face centroids of all the faces in a specified thread to a file. The `Get_Domain(1)` function call returns the pointer to the domain (or mixture domain, in the case of a multiphase application). This argument is not passed to `DEFINE_ON_DEMAND`.

```
/*******************************************************************
    Example of UDF for single phase that uses Get_Domain utility
*******************************************************************/

#include "udf.h"

FILE *fout;

void Print_Thread_Face_Centroids(Domain *domain, int id)
{
  real FC[2];
  face_t f;
  Thread *t = Lookup_Thread(domain, id);

  fprintf(fout,"thread id %d\n", id);
  begin_f_loop(f,t)
    {
     F_CENTROID(FC,f,t);
     fprintf(fout, "f%d %g %g %g\n", f, FC[0], FC[1], FC[2]);
    }
  end_f_loop(f,t)
  fprintf(fout, "\n");
}


DEFINE_ON_DEMAND(get_coords)
{
  Domain *domain;
  domain = Get_Domain(1);
  fout = fopen("faces.out", "w");
  Print_Thread_Face_Centroids(domain, 2);
  Print_Thread_Face_Centroids(domain, 4);
  fclose(fout);
}
```

Note that Get_Domain(1) replaces the extern Domain *domain expression used in previous releases of FLUENT 6.

Set Boundary Condition Value (`F_PROFILE`)

`F_PROFILE` is typically used in a `DEFINE_PROFILE` UDF to set a boundary condition value in memory for a given face and thread. The index `i` that is an argument to `F_PROFILE` is also an argument to `DEFINE_PROFILE` and identifies the particular boundary variable (e.g., pressure, temperature, velocity) that is to be set. `F_PROFILE` is defined in `mem.h`.

Macro:	`F_PROFILE(f, t, i)`

Argument types:	`face_t f`
	`Thread *t`
	`int i`

Function returns:	`void`

The arguments of `F_PROFILE` are `f`, the index of the face `face_t`; `t`, a pointer to the face's thread `t`; and `i`, an integer index to the particular face variable that is to be set. `i` is defined by ANSYS FLUENT when you hook a `DEFINE_PROFILE` UDF to a particular variable (e.g., pressure, temperature, velocity) in a boundary condition dialog box. This index is passed to your UDF by the ANSYS FLUENT solver so that the function knows which variable to operate on.

Suppose you want to define a custom inlet boundary pressure profile for your ANSYS FLUENT case defined by the following equation:

$$p(y) = 1.1 \times 10^5 - 0.1 \times 10^5 \left(\frac{y}{0.0745} \right)^2$$

You can set the pressure profile using a `DEFINE_PROFILE` UDF. Since a profile is an array of data, your UDF will need to create the pressure array by looping over all faces in the boundary zone, and for each face, set the pressure value using `F_PROFILE`. In the sample UDF source code shown below, the y coordinate of the centroid is obtained using `F_CENTROID`, and this value is used in the pressure calculation that is stored for each face. The solver passes the UDF the right index to the pressure variable because the UDF is hooked to Gauge Total Pressure in the Pressure Inlet boundary condition dialog box. See Section 2.3.15: `DEFINE_PROFILE` for more information on `DEFINE_PROFILE` UDFs.

```
/************************************************************************
   UDF for specifying a parabolic pressure profile boundary profile
*************************************************************************/

#include "udf.h"

DEFINE_PROFILE(pressure_profile,t,i)
{
  real x[ND_ND];                    /* this will hold the position vector */
  real y;
  face_t f;

  begin_f_loop(f,t)
    {
      F_CENTROID(x,f,t);
      y = x[1];
      F_PROFILE(f,t,i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
    }
  end_f_loop(f,t)
}
```

THREAD_SHADOW(t)

THREAD_SHADOW returns the face thread that is the shadow of Thread *t if it is one of a face/face-shadow pair that comprise a thin wall. It returns NULL if the boundary is not part of a thin wall and is often used in an if statement such as:

```
if (!NULLP(ts = THREAD_SHADOW(t)))
{
    /* Do things here using the shadow wall thread (ts)  */
}
```

3.2.7 Model-Specific Macros

DPM Macros

The macros listed in Tables 3.2.26–3.2.31 can be used to return `real` variables associated with the Discrete Phase Model (DPM), in SI units. They are typically used in DPM UDFs that are described in Section 2.5: Discrete Phase Model (DPM) DEFINE Macros. The variables are available in both the pressure-based and the density-based solver. The macros are defined in the `dpm.h` header file, which is included in `udf.h`.

The variable `p` indicates a pointer to the `Tracked_Particle` structure (`Tracked_Particle *p`) which gives you the value for the particle at the current position.

Refer to the following sections for examples of UDFs that utilize some of these macros: Section 2.5.7: DEFINE_DPM_LAW, Section 2.5.1: DEFINE_DPM_BC, Section 2.5.6: DEFINE_DPM_INJECTION_INIT, Section 2.5.13: DEFINE_DPM_SWITCH, and Section 2.5.9: DEFINE_DPM_PROPERTY.

Table 3.2.26: Macros for Particles at Current Position Defined in `dpm.h`

Macro	Argument Types	Returns
P_POS(p)[i]	Tracked_Particle *p int i	position i=0,1,2
P_VEL(p)[i]	Tracked_Particle *p int i	velocity i=0,1,2
P_DIAM(p)	Tracked_Particle *p	diameter
P_T(p)	Tracked_Particle *p	temperature
P_RHO(p)	Tracked_Particle *p	density
P_MASS(p)	Tracked_Particle *p	mass
P_TIME(p)	Tracked_Particle *p	current particle time
P_DT(p)	Tracked_Particle *p	time step
P_FLOW_RATE(p)	Tracked_Particle *p	flow rate of particles in a stream in kg/s (see below for details)
P_LF(p)	Tracked_Particle *p	liquid fraction (wet combusting particles only)
P_VFF(p)	Tracked_Particle *p	volatile fraction (combusting particles only)

`P_FLOW_RATE(p)`

Each particle in a steady flow calculation represents a "stream" of many particles that follow the same path. The number of particles in this stream that passes a particular point in a second is the "strength" of the stream. `P_FLOW_RATE` returns the strength multiplied by `P_MASS(p)` at the current particle position.

Table 3.2.27: Macros for Particles at Entry to Current Cell Defined in `dpm.h`

Macro	Argument Types	Returns
`P_POS0(p)[i]`	`Tracked_Particle *p int i`	position i=0,1,2
`P_VEL0(p)[i]`	`Tracked_Particle *p int i`	velocity i=0,1,2
`P_DIAM0(p)`	`Tracked_Particle *p`	diameter
`P_T0(p)`	`Tracked_Particle *p`	temperature
`P_RHO0(p)`	`Tracked_Particle *p`	density
`P_MASS0(p)`	`Tracked_Particle *p`	mass
`P_TIME0(p)`	`Tracked_Particle *p`	particle time at entry
`P_LF0(p)`	`Tracked_Particle *p`	liquid fraction (wet combusting particles only)

> **i** Note that when you are the using the macros listed in Table 3.2.27 to track transient particles, the particle state is the beginning of the fluid flow time step only if the particle does *not* cross a cell boundary.

Table 3.2.28: Macros for Particle Cell Index and Thread Pointer Defined in `dpm.h`

Name(Arguments)	Argument Types	Returns
`P_CELL(p)`	`Tracked_Particle *p`	cell index of the cell that the particle is currently in
`P_CELL_THREAD(p)`	`Tracked_Particle *p`	pointer to the thread of the cell that the particle is currently in

Table 3.2.29: Macros for Particles at Injection into Domain Defined in `dpm.h`

Macro	Argument Types	Returns
`P_INIT_POS(p)[i]`	`Tracked_Particle *p int i`	position i=0,1,2
`P_INIT_VEL(p)[i]`	`Tracked_Particle *p int i`	velocity i=0,1,2
`P_INIT_DIAM(p)`	`Tracked_Particle *p`	diameter
`P_INIT_TEMP(p)`	`Tracked_Particle *p`	temperature
`P_INIT_RHO(p)`	`Tracked_Particle *p`	density
`P_INIT_MASS(p)`	`Tracked_Particle *p`	mass
`P_INIT_LF(p)`	`Tracked_Particle *p`	liquid fraction (wet combusting particles only)

Table 3.2.30: Macros for Particle Species, Laws, and User Scalars Defined in `dpm.h`

Macro	Argument Types	Returns
`P_EVAP_SPECIES_INDEX(p)`	`Tracked_Particle *p`	evaporating species index in mixture
`P_DEVOL_SPECIES_INDEX(p)`	`Tracked_Particle *p`	devolatilizing species index in mixture.
`P_OXID_SPECIES_INDEX(p)`	`Tracked_Particle *p`	oxidizing species index in mixture
`P_PROD_SPECIES_INDEX(p)`	`Tracked_Particle *p`	combustion products species index in mixture
`P_CURRENT_LAW(p)`	`Tracked_Particle *p`	current particle law index
`P_NEXT_LAW(p)`	`Tracked_Particle *p`	next particle law index
`P_USER_REAL(p,i)`	`Tracked_Particle *p`	storage array for user-defined values (indexed by `i`)

Table 3.2.31: Macros for Particle Material Properties Defined in `dpm.h`

Macro	Argument Types	Returns
`P_MATERIAL(p)`	`Tracked_Particle *p`	material pointer
`DPM_BOILING_TEMPERATURE (p,m)`	`Tracked_Particle *p,` `Material *m`	boiling temperature
`DPM_CHAR_FRACTION(p)`	`Tracked_Particle *p`	char fraction
`DPM_DIFFUSION_COEFF(p,t)`	`Tracked_Particle *p,` particle temperature `t`	diffusion coefficient to be used the gaseous boundary layer around particle
`DPM_EMISSIVITY(p,m)`	`Tracked_Particle *p,` `Material *m`	emissivity for the radiation model
`DPM_EVAPORATION_`	`Tracked_Particle *p,` `TEMPERATURE(p,m)`	evaporation temperature
`DPM_HEAT_OF_PYROLYSIS(p)`	`Tracked_Particle *p`	heat of pyrolysis
`DPM_HEAT_OF_REACTION(p)`	`Tracked_Particle *p`	heat of reaction
`DPM_LATENT_HEAT(p)`	`Tracked_Particle *p`	latent heat
`DPM_LIQUID_SPECIFIC_HEAT (p,t)`	`Tracked_Particle *p,` particle temperature `t` Note: particle temp. typically determined by `P_T(p)`	specific heat of material used for liquid associated with particle
`DPM_MU(p)`	`Tracked_Particle *p`	dynamic viscosity of droplets
`DPM_SCATT_FACTOR(p,m)`	`Tracked_Particle *p,` `Material *m`	scattering factor for radiation model
`DPM_SPECIFIC_HEAT(p,t)`	`Tracked_Particle *p,` particle temperature `t` Note: particle temperature is typically determined by `P_T(p)`	specific heat at temperature `t`
`DPM_SWELLING_COEFF(p)`	`Tracked_Particle *p`	swelling coefficient for devolatilization
`DPM_SURFTEN(p)`	`Tracked_Particle *p`	surface tension of droplets
`DPM_VAPOR_PRESSURE(p,m)`	`Tracked_Particle *p,` `Material *m`	vapor pressure of liquid part of particle
`DPM_VAPOR_TEMP(p,m)`	`Tracked_Particle *p,` `Material *m`	vaporization temperature used to switch to vaporization law
`DPM_VOLATILE_FRACTION(p)`	`Tracked_Particle *p`	volatile fraction

NO$_x$ Macros

The following macros can be used in NO$_x$ model UDFs in the calculation of pollutant rates. These macros are defined in the header file `sg_nox.h`, which is included in `udf.h`. They can be used to return `real` NO$_x$ variables in SI units, and are available in both the pressure-based and the density-based solver. See Section 2.3.12: DEFINE_NOX_RATE for examples of DEFINE_NOX_RATE UDFs that utilize these macros.

Table 3.2.32: Macros for NO$_x$ UDFs Defined in `sg_nox.h`

Macro	Returns
POLLUT_EQN(Pollut_Par)	index of pollutant equation being solved (see below)
MOLECON(Pollut,SPE)	molar concentration of species specified by SPE (see below)
NULLIDX(Pollut_Par,SPE)	TRUE if the species specified by SPE doesn't exist in ANSYS FLUENT case (i.e., in the Species dialog box)
ARRH(Pollut,K)	Arrhenius rate calculated from the constants specified by K (see below)
POLLUT_FRATE(Pollut)	production rate of the pollutant species being solved
POLLUT_RRATE(Pollut)	reduction rate of the pollutant species being solved
POLLUT_QRATE(Pollut)	quasi-steady rate of N$_2$O formation (if the quasi-steady model is used)
POLLUT_FLUCTDEN(Pollut)	fluctuating density value (or, if no PDF model is used, mean density at a given cell
POLLUT_FLUCTTEM(Pollut)	fluctuating temperature value (or, if no PDF model is used, mean temperature at a given cell)
POLLUT_FLUCTYI(Pollut,SPE)	fluctuating mass fraction value (or, if no PDF model is used, mean mass fraction at a given cell) of the species given by index SPE
POLLUT_CTMAX(Pollut_Par)	upper limit for the temperature PDF integration (see below)

> **i** `Pollut_Par` is a pointer to the `Pollut_Parameter` data structure that contains auxiliary data common to all pollutant species and `NOx` is a pointer to the `NOx_Parameter` data structure that contains data specific to the NO$_x$ model.

- `POLLUT_EQN(Pollut_Par)` returns the index of the pollutant equation currently being solved. The indices are `EQ_NO` for NO, `EQ_HCN` for HCN, `EQ_N2O` for N_2O, and `EQ_NH3` for NH_3.

- `MOLECON(Pollut,SPE)` returns the molar concentration of a species specified by `SPE`, which is either the name of the species or `IDX(i)` when the species is a pollutant (like NO). `SPE` must be replaced by one of the following identifiers: `FUEL`, `O2`, `O`, `OH`, `H2O`, `N2`, `N`, `CH`, `CH2`, `CH3`, `IDX(NO)`, `IDX(N2O)`, `IDX(HCN)`, `IDX(NH3)`. For example, for O_2 molar concentration you should call `MOLECON(Pollut, O2)`, whereas for NO molar concentration the call should be `MOLECON(Pollut, IDX(NO))`. The identifier `FUEL` represents the fuel species as specified in the Fuel Species drop-down list under Prompt NO Parameters in the NOx Model dialog box.

- `ARRH(Pollut,K)` returns the Arrhenius rate calculated from the constants specified by `K`. `K` is defined using the `Rate_Const` data type and has three elements - A, B, and C. The Arrhenius rate is given in the form of

$$R = AT^B \exp(-C/T)$$

where T is the temperature.

Note that the units of K must be in m-gmol-J-s.

- `POLLUT_CTMAX(Pollut_Par)` can be used to modify the $T_{\max}$ value used as the upper limit for the integration of the temperature PDF (when temperature is accounted for in the turbulence interaction modeling). You must make sure not to put this macro under any conditions within the UDF (e.g., `IN_PDF` or `OUT_PDF`).

SO$_x$ Macros

The following macros can be used in SO$_x$ model UDFs in the calculation of pollutant rates. These macros are defined in the header file `sg_nox.h`, which is included in `udf.h`. They can be used to return `real` SO$_x$ variables in SI units and are available in both the pressure-based and the density-based solver. See Section 2.3.20: `DEFINE_SOX_RATE` for examples of `DEFINE_SOX_RATE` UDFs that utilize these macros.

Table 3.2.33: Macros for SO$_x$ UDFs Defined in `sg_nox.h`

Macro	Returns
POLLUT_EQN(Pollut_Par)	index of pollutant equation being solved (see below)
MOLECON(Pollut,SPE)	molar concentration of species specified by SPE (see below)
NULLIDX(Pollut_Par,SPE)	TRUE if the species specified by SPE doesn't exist in ANSYS FLUENT case (i.e., in the Species dialog box)
ARRH(Pollut,K)	Arrhenius rate calculated from the constants specified by K (see below)
POLLUT_FRATE(Pollut)	production rate of the pollutant species being solved
POLLUT_RRATE(Pollut)	reduction rate of the pollutant species being solved
POLLUT_FLUCTDEN(Pollut)	fluctuating density value (or, if no PDF model is used, mean density at a given cell)
POLLUT_FLUCTTEM(Pollut)	fluctuating temperature value (or, if no PDF model is used, mean temperature at a given cell)
POLLUT_FLUCTYI(Pollut,SPE)	fluctuating mass fraction value (or, if no PDF model is used, mean mass fraction at a given cell) of the species given by index SPE
POLLUT_CTMAX(Pollut_Par)	upper limit for the temperature PDF integration (see below)

> ℹ️ `Pollut_Par` is a pointer to the `Pollut_Parameter` data structure that contains auxiliary data common to all pollutant species and `SOx` is a pointer to the `SOx_Parameter` data structure that contains data specific to the SO$_x$ model.

- `POLLUT_EQN(Pollut_Par)` returns the index of the pollutant equation currently being solved. The indices are `EQ_SO2` for SO$_2$ and `EQ_SO3` for SO$_3$, etc.

- `MOLECON(Pollut, SPE)` returns the molar concentration of a species specified by SPE. SPE is either the name of the species or `IDX(i)` when the species is a pollutant (like SO$_2$). For example, for O$_2$ molar concentration you should call `MOLECON(Pollut, O2)`, whereas for SO$_2$ molar concentration the call should be `MOLECON(Pollut, IDX(SO2))`.

- `ARRH(Pollut,K)` returns the Arrhenius rate calculated from the constants specified by `K`. `K` is defined using the `Rate_Const` data type and has three elements - A, B, and C. The Arrhenius rate is given in the form of

$$R = AT^B \exp(-C/T)$$

  where T is the temperature.

  Note that the units of K must be in m-gmol-J-s.

- `POLLUT_CTMAX(Pollut_Par)` can be used to modify the $T_{\max}$ value used as the upper limit for the integration of the temperature PDF (when temperature is accounted for in the turbulence interaction modeling). You must make sure not to put this macro under any conditions within the UDF (e.g., `IN_PDF` or `OUT_PDF`).

Dynamic Mesh Macros

The macros listed in Table 3.2.34 are useful in dynamic mesh UDFs. The argument `dt` is a pointer to the dynamic thread structure, and `time` is a real value. These macros are defined in the `dynamesh_tools.h`.

Table 3.2.34: Macros for Dynamic Mesh Variables Defined in `dynamesh_tools.h`

Name(Arguments)	Argument Types	Returns
`DT_THREAD(dt)`	`Dynamic_Thread *dt`	pointer to a thread
`DT_CG(dt)`	`Dynamic_Thread *dt`	center of gravity vector
`DT_VEL_CG(dt)`	`Dynamic_Thread *dt`	cg velocity vector
`DT_OMEGA_CG(t)`	`Dynamic_Thread *dt`	angular velocity vector
`DT_THETA(dt)`	`Dynamic_Thread *dt`	orientation of body-fixed axis vector
`DYNAMESH_CURRENT_TIME`	N/A	current dynamic mesh time
`TIME_TO_ABSOLUTE_CRANK_ANGLE(time)`	real time	absolute value of the crank angle

See Section 2.6.4: `DEFINE_GRID_MOTION` for an example UDF that utilizes `DT_THREAD`.

3.2.8 User-Defined Scalar (UDS) Transport Equation Macros

This section contains macros that you can use when defining scalar transport UDFs in ANSYS FLUENT. Note that if you try to use the macros listed below (e.g., F_UDSI, C_UDSI) before you have specified user-defined scalars in your ANSYS FLUENT model (in the User-Defined Scalars dialog box), then an error will result.

Set_User_Scalar_Name

ANSYS FLUENT assigns a default name for every user-defined scalar that you allocate in the graphical user-interface. For example, if you specify 2 as the Number of User-Defined Scalars, then two variables with default names User Scalar 0 and User Scalar 1 will be defined and the variables with these default names will appear in setup and postprocessing dialog boxes. You can change the default names if you wish, using Set_User_Scalar_Name as described below.

The default name that appears in the graphical user interface and on plots in ANSYS FLUENT for user-defined scalars (e.g., User Scalar 0) can now be changed using the function Set_User_Scalar_Name.

```
void Set_User_Scalar_Name(int i,char *name);
```

i is the index of the scalar and name is a string containing the name you wish to assign. It is defined in sg_udms.h.

Set_User_Scalar_Name should be used only once and is best used in an EXECUTE_ON_LOADING UDF (see Section 2.2.6: DEFINE_EXECUTE_ON_LOADING). Due to the mechanism used, UDS variables cannot be renamed after they have been set, so if the name is changed in a UDF, for example, and the UDF library is reloaded, then the old name could remain. In this case, restart ANSYS FLUENT and load the library again.

F_UDSI

You can use F_UDSI when you want to access face variables that are computed for user-defined scalar transport equations (Table 3.2.35). See Section 3.2.9: Example UDF that Utilizes UDM and UDS Variables for an example of F_UDSI usage.

Table 3.2.35: Accessing User-Defined Scalar Face Variables (`mem.h`)

Macro	Argument Types	Returns
F_UDSI(f,t,i)	`face_t f, Thread *t, int i` Note: `i` is index of scalar	UDS face variables

| i | Note that F_UDSI is available for wall and flow boundary faces, only. If a UDS attempts to access any other face zone, then an error will result. |

C_UDSI

You can use C_UDSI when you want to access cell variables that are computed for user-defined scalar transport equations. Macros for accessing UDS cell variables are listed in Table 3.2.36. Some examples of usage for these macros include defining non-constant source terms for UDS transport equations and initializing equations. See Section 3.2.9: Example UDF that Utilizes UDM and UDS Variables for an example of C_UDSI usage.

Table 3.2.36: C_UDSI for Accessing UDS Transport Cell Variables (`mem.h`)

Macro	Argument Types	Returns
C_UDSI(c,t,i)	`cell_t c, Thread *t, int i`	UDS cell variables
C_UDSI_G(c,t,i)	`cell_t c, Thread *t, int i`	UDS gradient
C_UDSI_M1(c,t,i)	`cell_t c, Thread *t, int i`	UDS previous time step
C_UDSI_M2(c,t,i)	`cell_t c, Thread *t, int i`	UDS second previous time step
C_UDSI_DIFF(c,t,i)	`cell_t c, Thread *t, int i` Note: `i` is index of scalar	UDS diffusivity

Reserving UDS Variables

Reserve_User_Scalar_Vars

The new capability of loading more than one UDF library into ANSYS FLUENT raises the possibility of user-defined scalar (UDS) clashes. To avoid data contention between multiple UDF libraries using the same user-defined scalars, ANSYS FLUENT has provided the macro `Reserve_User_Scalar_Vars` that allows you to reserve scalars prior to use.

```
int Reserve_User_Scalar_Vars(int num)
```

`int num` is the number of user-defined scalars that the library uses. The integer returned is the lowest UDS index that the library may use. After calling:

```
offset = Reserve_User_Scalar_Vars(int num);
```

the library may safely use `C_UDSI(c,t,offset)` to `C_UDSI(c,t,offset+num-1)`. See Section 2.2.6: DEFINE_EXECUTE_ON_LOADING for an example of macro usage. Note that there are other methods you can use within UDFs to hardcode the offset to prevent data contention.

`Reserve_User_Scalar_Vars` (defined in `sg_udms.h`) is designed to be called from an EXECUTE_ON_LOADING UDF (Section 2.2.6: DEFINE_EXECUTE_ON_LOADING). An on-loading UDF, as its name implies, executes as soon as the shared library is loaded into AN-SYS FLUENT. The macro can also be called from an INIT or ON_DEMAND UDF. After a user scalar has bee reserved, it can be set to unique names for the particular library using `Set_User_Memory_Name` (see below for details on `Set_User_Memory_Name`). After the number of UDS that are needed by a particular library is set in the GUI and the variables are successfully reserved for the loaded library, the other functions in the library can safely use `C_UDMI(c,t,offset)` up to `C_UDMI(c,t,offset+num-1)` to store values in user scalars without interference.

Unreserving UDS Variables

ANSYS FLUENT does not currently provide the capability to unreserve UDS variables using a macro. `Unreserve` macros will be available in future versions of ANSYS FLUENT.

N_UDS

You can use `N_UDS` to access the number of user-defined scalar (UDS) transport equations that have been specified in ANSYS FLUENT. The macro takes no arguments and returns the integer number of equations. It is defined in `models.h`.

3.2.9 User-Defined Memory (UDM) Macros

This section contains macros that access user-defined memory (UDM) variables in ANSYS FLUENT.

Before you can store variables in memory using the macros provided below, you will first need to allocate the appropriate number of memory location(s) in the User-Defined Memory dialog box in ANSYS FLUENT. (See Section 6.1.8: User-Defined Memory Storage for more details.)

Define ⟶ User-Defined ⟶ Memory...

> *i* Note that if you try to use F_UDMI or C_UDMI before you have allocated memory, then an error will result.

A variable will be created for every user-defined memory location that you allocate in the graphical user-interface. For example, if you specify 2 as the Number of User-Defined Memory, then two variables with default names User Memory 0 and User Memory 1 will be defined for your model and the default variable names will appear in postprocessing dialog boxes. You can change the default names if you wish, using Set_User_Memory_Name as described below.

Set_User_Memory_Name

The default name that appears in the graphical user interface and on plots for user-defined memory (UDM) values in ANSYS FLUENT (e.g., User Memory 0) can now be changed using the function Set_User_Memory_Name.

```
void Set_User_Memory_Name(int i,char *name);
```

i is the index of the memory value and name is a string containing the name you wish to assign. It is defined in sg_udms.h.

The Set_User_Memory_Name function should be used only once and is best used in an EXECUTE_ON_LOADING UDF (see Section 2.2.6: DEFINE_EXECUTE_ON_LOADING). Due to the mechanism used, User Memory values cannot be renamed after they have been set, so if the name is changed in a UDF, for example, and the UDF library is reloaded, then the old name could remain. In this case, restart ANSYS FLUENT and load the library again.

F_UDMI

You can use F_UDMI (Table 3.2.37) to access or store the value of the user-defined memory on a face. F_UDMI can be used to allocate up to 500 memory locations in order to store and retrieve the values of face field variables computed by UDFs. These stored values can then be used for postprocessing, for example, or by other UDFs.

i Note that F_UDMI is available for wall and flow boundary faces, only.

Table 3.2.37: Storage of User-Defined Memory on Faces (`mem.h`)

Macro	Argument Types	Usage
F_UDMI(f,t,i)	face_t f, Thread *t, int i	stores the face value of a user-defined memory with index i

There are three arguments to F_UDMI: f, t, and i. f is the face identifier, t is a pointer to the face thread, and i is an integer index that identifies the memory location where data is to be stored. An index i of 0 corresponds to user-defined memory location 0 (or User Memory 0).

Example

```
/* Compute face temperature and store in user-defined memory  */
begin_f_loop(f,t)
  {
    temp = F_T(f,t);
    F_UDMI(f,t,0) = (temp - tmin) / (tmax-tmin);
  }
end_f_loop(f,t)
}
```

See Section 2.5.4: DEFINE_DPM_EROSION for another example of F_UDMI usage.

```
C_UDMI
```

You can use C_UDMI to access or store the value of the user-defined memory in a cell. C_UDMI can be used to allocate up to 500 memory locations in order to store and retrieve the values of cell field variables computed by UDFs (Table 3.2.38). These stored values can then be used for postprocessing, for example, or by other UDFs. See Section 3.2.9: Example UDF that Utilizes UDM and UDS Variables for an example of C_UDMI usage.

Table 3.2.38: Storage of User-Defined Memory in Cells (`mem.h`)

Macro	Argument Types	Usage
C_UDMI(c,t,i)	cell_t c, Thread *t, int i	stores the cell value of a user-defined memory with index i

There are three arguments to C_UDMI: c, thread, and i. c is the cell identifier, thread is a pointer to the cell thread, and i is an integer index that identifies the memory location where data is to be stored. An index i of 0 corresponds to user-defined memory location 0 (or User Memory 0).

Example UDF that Utilizes UDM and UDS Variables

UDMs are often used to store diagnostic values derived from calculated values of a UDS. Below is an example that shows a technique for plotting the gradient of any flow variable. In this case, the volume fraction of a phase is loaded into a user scalar. If an iteration is made such that the UDS is not calculated, the gradients of the scalar will nevertheless be updated without altering the values of the user scalar. The gradient is then available to be copied into a User Memory variable for displaying.

```
# include "udf.h"
# define domain_ID 2

DEFINE_ADJUST(adjust_gradient, domain)
{
  Thread *t;
  cell_t c;
  face_t f;

  domain = Get_Domain(domain_ID);

  /* Fill UDS with the variable. */
  thread_loop_c (t,domain)
```

```
      {
         begin_c_loop (c,t)
           {
              C_UDSI(c,t,0) = C_VOF(c,t);
           }
         end_c_loop (c,t)
      }

   thread_loop_f (t,domain)
      {
         if (THREAD_STORAGE(t,SV_UDS_I(0))!=NULL)
         begin_f_loop (f,t)
           {
              F_UDSI(f,t,0) = F_VOF(f,t);
           }
         end_f_loop (f,t)
      }
 }

 DEFINE_ON_DEMAND(store_gradient)
 {
   Domain *domain;
   cell_t c;
   Thread *t;

   domain=Get_Domain(1);

   /* Fill the UDM with magnitude of gradient. */
   thread_loop_c (t,domain)
      {
         begin_c_loop (c,t)
           {
              C_UDMI(c,t,0) = NV_MAG(C_UDSI_G(c,t,0));
           }
         end_c_loop (c,t)
      }
 }
```

Reserving UDM Variables Using `Reserve_User_Memory_Vars`

The capability of loading more than one UDF library into ANSYS FLUENT raises the possibility of user-defined memory (UDM) clashes. If, for example, you want to use one UDF library that has a fixed 2D magnetic field stored in `User Memory 0` and `User Memory 1` and you want to use another UDF library that models the mass exchange between phases using `User Memory 0` for the exchange rates and these two libraries are loaded at the same time, then the two models are going to interfere with each other's data in `User Memory 0`. To avoid data contention problems, ANSYS FLUENT has a macro that will allow a UDF library to "reserve" UDM locations prior to usage. Note that there are other methods you can use within UDFs to hardcode the offset for UDMs to prevent contention that are not discussed here.

```
int Reserve_User_Memory_Vars(int num)
```

The integer given as an argument to the macro (`num`) specifies the number of UDMs needed by the library. The integer returned by the function is the starting point or "offset" from which the library may use the UDMs. It should be saved as a global integer such as `offset` in the UDF and it should be initialized to the special variable `UDM_UNRESERVED`.

```
offset = Reserve_User_Memory_Vars(int num);
```

`Reserve_User_Memory_Vars` (defined in `sg_udms.h`) is designed to be called from an `EXECUTE_ON_LOADING` UDF (Section 2.2.6: `DEFINE_EXECUTE_ON_LOADING`). An on-loading UDF, as its name implies, executes as soon as the shared library is loaded into ANSYS FLUENT. The macro can also be called from an `INIT` or `ON_DEMAND` UDF, although this is discouraged except for testing purposes. After a UDM is reserved, it can be set to unique names for the particular library using `Set_User_Memory_Name` (see below for details.) After the number of UDMs that are needed by a particular library is set in the GUI and the UDMs are successfully reserved for the loaded library, the other functions in the library can safely use `C_UDMI(c,t,offset)` up to `C_UDMI(c,t,offset+num-1)` to store values in memory locations without interference. Two example source code files named `udm_res1.c` and `udm_res2.c` each containing two UDFs are listed below. The first UDF is an `EXECUTE_ON_LOADING` UDF that is used to reserve UDMs for the library and set unique names for the UDM locations so that they can be easily identified in postprocessing. The second UDF is an `ON_DEMAND` UDF that is used to set the values of the UDM locations after the solution has been initialized. The `ON_DEMAND` UDF sets the initial values of the UDM locations using `udf_offset`, which is defined in the `EXECUTE_ON_LOADING` UDF. Note that the on demand UDF must be executed *after* the solution is initialized to reset the initial values for the UDMs.

The following describes the process of reserving five UDMs for two libraries named `libudf` and `libudf2`.

1. In the User-Defined Memory dialog box, specify 5 for the Number of User-Defined Memory Locations.

2. In the Compiled UDFs dialog box, build the compiled library named `libudf` for `udm_res1.c` and load the library.

3. Build the compiled library for `udm_res2.c` named `libudf2` and load the library.

4. Initialize the solution.

5. Execute the `on-demand` UDFs for `libudf` and `libudf2` in the Execute On Demand dialog box.

6. Iterate the solution.

7. Postprocess the results.

Example 1

```
/*************************************************************************
udm_res1.c contains two UDFs: an execute on loading UDF that reserves
three UDMs for libudf and renames the UDMs to enhance postprocessing,
and an on-demand UDF that sets the initial value of the UDMs.
*************************************************************************/
#include "udf.h"

#define NUM_UDM 3
static int udm_offset = UDM_UNRESERVED;

DEFINE_EXECUTE_ON_LOADING(on_loading, libname)
{
  if (udm_offset == UDM_UNRESERVED) udm_offset =
          Reserve_User_Memory_Vars(NUM_UDM);

  if (udm_offset == UDM_UNRESERVED)
     Message("\nYou need to define up to %d extra UDMs in GUI and "
             "then reload current library %s\n", NUM_UDM, libname);
  else
        {
  Message("%d UDMs have been reserved by the current "
              "library %s\n",NUM_UDM, libname);
```

```
  Set_User_Memory_Name(udm_offset,"lib1-UDM-0");
  Set_User_Memory_Name(udm_offset+1,"lib1-UDM-1");
          Set_User_Memory_Name(udm_offset+2,"lib1-UDM-2");
}
  Message("\nUDM Offset for Current Loaded Library = %d",udm_offset);
}


DEFINE_ON_DEMAND(set_udms)
{
  Domain *d;
  Thread *ct;
  cell_t c;
  int i;

  d=Get_Domain(1);

  if(udm_offset != UDM_UNRESERVED)
          {
          Message("Setting UDMs\n");

          for (i=0;i<NUM_UDM;i++)
                  {
                  thread_loop_c(ct,d)
                          {
                          begin_c_loop(c,ct)
                             {
                              C_UDMI(c,ct,udm_offset+i)=3.0+i/10.0;
                             }
                          end_c_loop(c,ct)
                          }
                  }
          }
  else
     Message("UDMs have not yet been reserved for library 1\n");
}
```

Example 2

```
*********************************************************************/
udm_res2.c contains two UDFs: an execute on loading UDF that reserves
two UDMs for libudf and renames the UDMs to enhance postprocessing,
and an on-demand UDF that sets the initial value of the UDMs.
*********************************************************************/
#include "udf.h"

#define NUM_UDM 2
static int udm_offset = UDM_UNRESERVED;

DEFINE_EXECUTE_ON_LOADING(on_loading, libname)
{
  if (udm_offset == UDM_UNRESERVED) udm_offset =
           Reserve_User_Memory_Vars(NUM_UDM);

  if (udm_offset == UDM_UNRESERVED)
      Message("\nYou need to define up to %d extra UDMs in GUI and "
              "then reload current library %s\n", NUM_UDM, libname);
  else
        {
  Message("%d UDMs have been reserved by the current "
              "library %s\n",NUM_UDM, libname);

  Set_User_Memory_Name(udm_offset,"lib2-UDM-0");
  Set_User_Memory_Name(udm_offset+1,"lib2-UDM-1");
}
  Message("\nUDM Offset for Current Loaded Library = %d",udm_offset);
}

DEFINE_ON_DEMAND(set_udms)
{
  Domain *d;
  Thread *ct;
  cell_t c;
  int i;

  d=Get_Domain(1);

  if(udm_offset != UDM_UNRESERVED)
        {
          Message("Setting UDMs\n");
```

```
        for (i=0;i<NUM_UDM;i++)
            {
            thread_loop_c(ct,d)
                {
                begin_c_loop(c,ct)
                    {
                     C_UDMI(c,ct,udm_offset+i)=2.0+i/10.0;
                    }
                end_c_loop(c,ct)
                }
            }
        }
  else
     Message("UDMs have not yet been reserved for library 1\n");
}
```

If your model uses a number of UDMs, it may be useful to define your variables in an easy-to-read format, either at the top of the source file or in a separate header file using the preprocessor `#define` directive:

```
    #define C_MAG_X(c,t)C_UDMI(c,t,udm_offset)
    #define C_MAG_Y(c,t)C_UDMI(c,t,udm_offset+1)
```

Following this definition, in the remainder of your UDF you can simply use `C_MAG_X(c,t)` and `C_MAG_Y(c,t)` to specify the fixed magnetic field components.

Unreserving UDM variables

ANSYS FLUENT does not currently provide the capability to unreserve UDM variables using a macro. `Unreserve` macros will be available in future versions of ANSYS FLUENT. You will need to exit ANSYS FLUENT to ensure that all UDM variables are reset.

3.3 Looping Macros

Many UDF tasks require repeated operations to be performed on nodes, cells, and threads in a computational domain. For your convenience, ANSYS FLUENT has provided you with a set of predefined macros to accomplish.looping tasks. For example, to define a custom boundary profile function you will need to loop over all the faces in a face thread using `begin..end_f_loop` looping macros. For operations where you want to loop over all the faces or cells in a domain, you will need to nest a `begin..end_f_loop` or `begin..end_c_loop` inside a `thread_loop_f` or `thread_loop_c`, respectively.

The following general looping macros can be used for UDFs in single-phase or multiphase models in ANSYS FLUENT. Definitions for these macros are contained in the `mem.h` header file.

> i You should not access a scheme variable using any of the `RP_GET_...` functions from inside a cell or face looping macro (c_loop or f_loop). This type of communication between the solver and cortex is very time consuming and therefore should be done outside of loops.

Looping Over Cell Threads in a Domain (`thread_loop_c`)

You can use `thread_loop_c` when you want to loop over all cell threads in a given domain. It consists of a single statement, followed by the operation(s) to be performed on all cell threads in the domain enclosed within braces `{}` as shown below. Note that `thread_loop_c` is similar in implementation to the `thread_loop_f` macro described below.

```
Domain *domain;
Thread *c_thread;
thread_loop_c(c_thread, domain) /*loops over all cell threads in domain*/
  {
  }
```

Looping Over Face Threads in a Domain (`thread_loop_f`)

You can use `thread_loop_f` when you want to loop over all face threads in a given domain. It consists of a single statement, followed by the operation(s) to be performed on all face threads in the domain enclosed within braces `{}` as shown below. Note that `thread_loop_f` is similar in implementation to the `thread_loop_c` macro described above.

```
Thread *f_thread;
Domain *domain;
thread_loop_f(f_thread, domain)/* loops over all face threads in a domain*/
  {
  }
```

Looping Over Cells in a Cell Thread (`begin...end_c_loop`)

You can use `begin_c_loop` and `end_c_loop` when you want to loop over all cells in a given cell thread. It contains a `begin` and `end` loop statement, and performs operation(s) on each cell in the cell thread as defined between the braces `{}`. This loop is usually nested within `thread_loop_c` when you want to loop over all cells in all cell threads in a domain.

```
cell_t c;
Thread *c_thread;
begin_c_loop(c, c_thread)    /* loops over cells in a cell thread  */
  {
  }
end_c_loop(c, c_thread)
```

Example

```
/* Loop over cells in a thread to get information stored in cells. */
  begin_c_loop(c, c_thread)
    {
     /* C_T gets cell temperature. The += will cause all of the cell
        temperatures to be added together. */

      temp += C_T(c, c_thread);
    }
  end_c_loop(c, c_thread)
}
```

Looping Over Faces in a Face Thread (`begin...end_f_loop`)

You can use `begin_f_loop` and `end_f_loop` when you want to loop over all faces in a given face thread. It contains a `begin` and `end` loop statement, and performs operation(s) on each face in the face thread as defined between the braces `{}`. This loop is usually nested within `thread_loop_f` when you want to loop over all faces in all face threads in a domain.

```
face_t f;
Thread *f_thread;
begin_f_loop(f, f_thread)    /* loops over faces in a face thread  */
  {
  }
end_f_loop(f, f_thread)
```

Example

```
/* Loop over faces in a face thread to get the information stored on faces. */
  begin_f_loop(f, f_thread)
    {
      /*  F_T gets face temperature. The += will cause all of the face
          temperatures to be added together. */

      temp += F_T(f, f_thread);
    }
  end_f_loop(f, f_thread)
```

Looping Over Faces of a Cell (c_face_loop)

The following looping function loops over all faces of a given cell. It consists of a single loop statement, followed by the action to be taken in braces {}.

```
cell_t c;
Thread *t;
face_t f;
Thread *tf;
int n;
c_face_loop(c, t, n)            /* loops over all faces of a cell */
  {
  .
  .
  .
  f = C_FACE(c,t,n);
  tf = C_FACE_THREAD(c,t,n);
  .
  .
  .
  }
```

The argument n is the local face index number. The local face index number is used in the C_FACE macro to obtain the global face number (e.g., f = C_FACE(c,t,n)).

Another useful macro that is often used in c_face_loop is C_FACE_THREAD. This macro is used to reference the associated face thread (e.g., tf = C_FACE_THREAD(c,t,n)).

Refer to Section 3.8: Miscellaneous Macros for other macros that are associated with c_face_loop.

Looping Over Nodes of a Cell (`c_node_loop`)

`c_node_loop(c,t,n)` is a function that loops over all nodes of a given cell. It consists of a single loop statement, followed by the action to be taken in braces {}.

Example:

```
cell_t c;
Thread *t;
int n;
Node *node;
c_node_loop(c,t,n)
  {
  .
  .
  node = C_NODE(c,t,n);
  .
  .
  }
```

Here, `n` is the local node index number. The index number can be used with the `C_NODE` macro to obtain the global cell node number (e.g., `node = C_NODE(c,t,n)`).

Looping Over Nodes of a Face (`f_node_loop`)

`f_node_loop(f,t,n)` is a function that loops over all nodes of a given face. It consists of a single loop statement, followed by the action to be taken in braces {}.

Example

```
face_t f;
Thread *t;
int n;
Node *node;
f_node_loop(f,t,n)
  {
  .
  .
  .
  node = F_NODE(f,t,n);
  .
  .
  .
  }
```

Here, `n` is the local node index number. The index number can be used with the `F_NODE` macro to obtain the global face node number (e.g., `node = F_NODE(f,t,n)`).

See Section 2.6.4: `DEFINE_GRID_MOTION` for an example of a UDF that uses `f_node_loop`.

3.3.1   Multiphase Looping Macros

This section contains a description of looping macros that are to be used for multiphase UDFs only. They enable your function to loop over all cells and faces for given threads or domains. Refer to Section 1.10.1: Multiphase-specific Data Types and, in particular, Figure 1.10.1 for a discussion on hierarchy of structures within ANSYS FLUENT.

Looping Over Phase Domains in Mixture (`sub_domain_loop`)

The `sub_domain_loop` macro loops over all phase domains (subdomains) within the mixture domain. The macro steps through and provides each phase domain pointer defined in the mixture domain as well as the corresponding `phase_domain_index`. As discussed in Section 1.10.1: Multiphase-specific Data Types, the domain pointer is needed, in part, to gain access to data within each phase. Note that `sub_domain_loop` is similar in implementation to the `sub_thread_loop` macro described below.

```
int phase_domain_index;    /* index of subdomain pointers  */
Domain *mixture_domain;
Domain *subdomain;
sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
```

The variable arguments to `sub_domain_loop` are `subdomain`, `mixture_domain`, and `phase_domain_index`. `subdomain` is a pointer to the phase-level domain, and `mixture_domain` is a pointer to the mixture-level domain. The `mixture_domain` is automatically passed to your UDF by the ANSYS FLUENT solver when you use a DEFINE macro that contains a domain variable argument (e.g., `DEFINE_ADJUST`) and your UDF is hooked to the mixture. If `mixture_domain` is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (e.g., `Get_Domain(1)`) before calling `sub_domain_loop` (see Section 3.2.6: Domain Pointer (`Get_Domain`)). `phase_domain_index` is an index of subdomain pointers. `phase_domain_index` is 0 for the primary phase, and is incremented by one for each secondary phase in the mixture. Note that `subdomain` and `phase_domain_index` are set within the `sub_domain_loop` macro.

Example

The following interpreted UDF patches an initial volume fraction for a particular phase in a solution. It is executed once at the beginning of the solution process. The function sets up a spherical volume centered at `0.5, 0.5, 0.5` with a radius of `0.25`. A secondary-phase volume fraction of `1` is then patched to the cells within the spherical volume, while the volume fraction for the secondary phase in all other cells is set to `0`.

```
/*******************************************************************
   UDF for initializing phase volume fraction
*******************************************************************/

#include "udf.h"

/* domain pointer that is passed by INIT function is mixture domain  */
DEFINE_INIT(my_init_function, mixture_domain)
{
  int phase_domain_index;
  cell_t cell;
  Thread *cell_thread;
  Domain *subdomain;
  real xc[ND_ND];

  /* loop over all subdomains (phases) in the superdomain (mixture) */
  sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
  {
     /* loop if secondary phase  */
     if (DOMAIN_ID(subdomain) == 3)

     /* loop over all cell threads in the secondary phase domain  */
     thread_loop_c (cell_thread,subdomain)
     {
         /* loop over all cells in secondary phase cell threads  */
         begin_c_loop_all (cell,cell_thread)
         {
             C_CENTROID(xc,cell,cell_thread);
             if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                             pow(xc[1] - 0.5,2.),
                             pow(xc[2] - 0.5,2.))) < 0.25)

               /*  set volume fraction to 1 for centroid  */
               C_VOF(cell,cell_thread) = 1.;
             else
               /*  otherwise initialize to zero  */
               C_VOF(cell,cell_thread) = 0.;
         }
         end_c_loop_all (cell,cell_thread)
     }

  }
}
```

Looping Over Phase Threads in Mixture (`sub_thread_loop`)

The `sub_thread_loop` macro loops over all phase-level threads (subthreads) associated with a mixture-level thread. The macro steps through and returns the pointer to each subthread as well as the corresponding `phase_domain_index`. As discussed in Section 1.10.1: Multiphase-specific Data Types, if the subthread pointer is associated with an inlet zone, then the macro will provide the pointers to the face threads associated with the inlet for each of the phases.

```
int phase_domain_index;
Thread *subthread;
Thread *mixture_thread;
sub_thread_loop(subthread, mixture_thread, phase_domain_index)
```

The variable arguments to `sub_thread_loop` are `subthread`, `mixture_thread`, and `phase_domain_index`. `subthread` is a pointer to the phase thread, and `mixture_thread` is a pointer to the mixture-level thread. The `mixture_thread` is automatically passed to your UDF by the ANSYS FLUENT solver when you use a DEFINE macro that contains a thread variable argument (e.g., DEFINE_PROFILE) and your UDF is hooked to the mixture. If the `mixture_thread` is not explicitly passed to your UDF, you will need to use a utility macro to retrieve it before calling `sub_thread_loop`. `phase_domain_index` is an index of subdomain pointers that can be retrieved using the PHASE_DOMAIN_INDEX macro. (See Section 3.3.2: Phase Domain Index (PHASE_DOMAIN_INDEX) for details.) The index begins at 0 for the primary phase, and is incremented by one for each secondary phase in the mixture. Note that `subthread` and `phase_domain_index` are initialized within the `sub_thread_loop` macro definition.

Looping Over Phase Cell Threads in Mixture (`mp_thread_loop_c`)

The `mp_thread_loop_c` macro loops through all cell threads (at the mixture level) within the mixture domain and provides the pointers of the phase-level (cell) threads associated with each mixture-level thread. This is nearly identical to the `thread_loop_c` macro (Section 3.3: Looping Over Cell Threads in a Domain (`thread_loop_c`)) when applied to the mixture domain. The difference is that, in addition to stepping through each cell thread, the macro also returns a pointer array (`pt`) that identifies the corresponding phase-level threads. The pointer to the cell thread for the ith phase is `pt[i]`, where `i` is the `phase_domain_index`. `pt[i]` can be used as an argument to macros requiring the phase-level thread pointer. `phase_domain_index` can be retrieved using the PHASE_DOMAIN_INDEX macro. (See Section 3.3.2: Phase Domain Index (PHASE_DOMAIN_INDEX) for details.)

```
Thread **pt;
Thread *cell_threads;
Domain *mixture_domain;
mp_thread_loop_c(cell_threads, mixture_domain, pt)
```

The variable arguments to `mp_thread_loop_c` are `cell_threads`, `mixture_domain`, and `pt`. `cell_threads` is a pointer to the cell threads, and `mixture_domain` is a pointer to the mixture-level domain. `pt` is an array pointer whose elements contain pointers to phase-level threads.

`mixture_domain` is automatically passed to your UDF by the ANSYS FLUENT solver when you use a `DEFINE` macro that contains a domain variable argument (e.g., `DEFINE_ADJUST`) and your UDF is hooked to the mixture. If `mixture_domain` is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (e.g., `Get_Domain(1)`, described in Section 3.2.6: Domain Pointer (`Get_Domain`)). Note that the values for `pt` and `cell_threads` are set within the looping function.

`mp_thread_loop_c` is typically used along with `begin_c_loop`. `begin_c_loop` loops over cells in a cell thread. When `begin_c_loop` is nested within `mp_thread_loop_c`, you can loop over all cells in all phase cell threads within a mixture.

Looping Over Phase Face Threads in Mixture (`mp_thread_loop_f`)

The `mp_thread_loop_f` macro loops through all face threads (at the mixture level) within the mixture domain and provides the pointers of the phase-level (face) threads associated with each mixture-level thread. This is nearly identical to the `thread_loop_f` macro when applied to the mixture domain. The difference is that, in addition to stepping through each face thread, the macro also returns a pointer array (`pt`) that identifies the corresponding phase-level threads. The pointer to the face thread for the ith phase is `pt[i]`, where i is the `phase_domain_index`. `pt[i]` can be used as an argument to macros requiring the phase-level thread pointer. The `phase_domain_index` can be retrieved using the `PHASE_DOMAIN_INDEX` macro. (See Section 3.3.2: Phase Domain Index (`PHASE_DOMAIN_INDEX`) for details.)

```
Thread **pt;
Thread *face_threads;
Domain *mixture_domain;
mp_thread_loop_f(face_threads, mixture_domain, pt)
```

The variable arguments to `mp_thread_loop_f` are `face_threads`, `mixture_domain`, and `pt`. `face_threads` is a pointer to the face threads, and `mixture_domain` is a pointer to the mixture-level domain. `pt` is an array pointer whose elements contain pointers to phase-level threads.

`mixture_domain` is automatically passed to your UDF by the ANSYS FLUENT solver if you are using a `DEFINE` macro that contains a domain variable argument (e.g., `DEFINE_ADJUST`) and your UDF is hooked to the mixture. If `mixture_domain` is not explicitly passed to your UDF, you may use another utility macro to retrieve it (e.g., `Get_Domain(1)`, described in Section 3.2.6: Domain Pointer (`Get_Domain`)). Note that the values for `pt` and `face_threads` are set within the looping function.

`mp_thread_loop_f` is typically used along with `begin_f_loop`. `begin_f_loop` loops over faces in a face thread. When `begin_f_loop` is nested within `mp_thread_loop_f`, you can loop over all faces in all phase face threads within a mixture.

3.3.2   Advanced Multiphase Macros

For most standard UDFs written for multiphase models (e.g., source term, material property, profile functions), variables that your function needs (domain pointers, thread pointers, etc.) are passed directly to your UDF as arguments by the solver in the solution process. All you need to do is hook the UDF to your model and everything is taken care of. For example, if your multiphase UDF defines a custom profile for a particular boundary zone (using `DEFINE_PROFILE`) and is hooked to the appropriate phase or mixture in ANSYS FLUENT in the relevant boundary condition dialog box, then appropriate phase or mixture variables will be passed to your function by the solver at run-time.

There may, however, be more complex functions you wish to write that require a variable that is *not* directly passed through its arguments. `DEFINE_ADJUST` and `DEFINE_INIT` functions, for example, are passed mixture domain variables only. If a UDF requires a phase domain pointer, instead, then it will need to utilize macros presented in this section to retrieve it. `ON_DEMAND` UDFS aren't directly passed any variables through their arguments. Consequently, any on demand function that requires access to phase or domain variables will also need to utilize macros presented in this section to retrieve them.

Recall that when you are writing UDFs for multiphase models, you will need to keep in mind the hierarchy of structures within ANSYS FLUENT (see Section 1.10.1: Multiphase-specific Data Types for details). The particular domain or thread structure that gets passed into your UDF from the solver depends on the `DEFINE` macro you are using, as well as the domain the function is hooked to (either through the graphical user interface, or hardwired in the code). As mentioned above, it also may depend on the multiphase model that you are using. Refer to Section 1.10.1: Multiphase-specific Data Types and, in particular, Figure 1.10.1 for a discussion on hierarchy of structures within ANSYS FLUENT.

Phase Domain Pointer (`DOMAIN_SUB_DOMAIN`)

There are two ways you can get access to a specific phase (or subdomain) pointer within the mixture domain. You can use either the `DOMAIN_SUB_DOMAIN` macro (described below) or `Get_Domain`, which is described below.

`DOMAIN_SUB_DOMAIN` has two arguments: `mixture_domain` and `phase_domain_index`. The function returns the phase pointer `subdomain` for the given `phase_domain_index`. Note that `DOMAIN_SUB_DOMAIN` is similar in implementation to the `THREAD_SUB_THREAD` macro described in Section 3.3.2: Phase-Level Thread Pointer (`THREAD_SUB_THREAD`).

```
int phase_domain_index = 0;                 /* primary phase index is 0 */
Domain *mixture_domain;
Domain *subdomain = DOMAIN_SUB_DOMAIN(mixture_domain,phase_domain_index);
```

`mixture_domain` is a pointer to the mixture-level domain. It is automatically passed to your UDF by the ANSYS FLUENT solver when you use a `DEFINE` macro that contains a domain variable argument (e.g., `DEFINE_ADJUST`) and your UDF is hooked to the mixture. Otherwise, if the `mixture_domain` is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (e.g., `Get_Domain(1)`) before calling `sub_domain_loop`.

`phase_domain_index` is an index of subdomain pointers. It is an integer that starts with `0` for the primary phase and is incremented by one for each secondary phase. `phase_domain_index` is automatically passed to your UDF by the ANSYS FLUENT solver when you use a `DEFINE` macro that contains a phase domain index argument (`DEFINE_EXCHANGE_PROPERTY`, `DEFINE_VECTOR_EXCHANGE_PROPERTY`) and your UDF is hooked to a specific interaction phase. Otherwise, you will need to hard code the integer value of `phase_domain_index` to the `DOMAIN_SUB_DOMAIN` macro. If your multiphase model has only two phases defined, then `phase_domain_index` is `0` for the primary phase, and `1` for the secondary phase. However, if you have more than one secondary phase defined for your multiphase model, you will need to use the `PHASE_DOMAIN_INDEX` utility to retrieve the corresponding `phase_domain_index` for the given domain. See Section 3.3.2: Phase Domain Index (`PHASE_DOMAIN_INDEX`) for details.

Phase-Level Thread Pointer (THREAD_SUB_THREAD)

The THREAD_SUB_THREAD macro can be used to retrieve the phase-level thread (sub-thread) pointer, given the phase domain index. THREAD_SUB_THREAD has two arguments: mixture_thread and phase_domain_index. The function returns the phase-level thread pointer for the given phase_domain_index. Note that THREAD_SUB_THREAD is similar in implementation to the DOMAIN_SUB_DOMAIN macro described in Section 3.3.2: Phase Domain Pointer (DOMAIN_SUB_DOMAIN).

```
int phase_domain_index = 0;            /* primary phase index is 0  */
Thread *mixture_thread;                /* mixture-level thread pointer */
Thread *subthread = THREAD_SUB_THREAD(mixture_thread,phase_domain_index);
```

mixture_thread is a pointer to a mixture-level thread. It is automatically passed to your UDF by the ANSYS FLUENT solver when you use a DEFINE macro that contains a variable thread argument (e.g., DEFINE_PROFILE), and the function is hooked to the mixture. Otherwise, if the mixture thread pointer is not explicitly passed to your UDF, then you will need to use the Lookup_Thread utility macro to retrieve it (see Section 3.2.6: Thread Pointer for Zone ID (Lookup_Thread)).

phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase. phase_domain_index is automatically passed to your UDF by the ANSYS FLUENT solver when you use a DEFINE macro that contains a phase domain index argument (DEFINE_EXCHANGE_PROPERTY, DEFINE_VECTOR_EXCHANGE_PROPERTY) and your UDF is hooked to a specific interaction phase. (See Section 2.4.2: DEFINE_EXCHANGE_PROPERTY for an example UDF.) Otherwise, you will need to hard code the integer value of phase_domain_index to the THREAD_SUB_THREAD macro. If your multiphase model has only two phases defined, then phase_domain_index is 0 for the primary phase, and 1 for the secondary phase. However, if you have more than one secondary phase defined for your multiphase model, you will need to use the PHASE_DOMAIN_INDEX utility to retrieve the corresponding phase_domain_index for the given domain. See Section 3.3.2: Phase Domain Index (PHASE_DOMAIN_INDEX) for details.

Phase Thread Pointer Array (`THREAD_SUB_THREAD`)

The `THREAD_SUB_THREADS` macro can be used to retrieve the pointer array, `pt`, whose elements contain pointers to phase-level threads (subthreads). `THREADS_SUB_THREADS` has one argument, `mixture_thread`.

```
Thread *mixture_thread;
Thread **pt;   /* initialize pt   */
pt = THREAD_SUB_THREADS(mixture_thread);
```

`mixture_thread` is a pointer to a mixture-level thread which can represent a cell thread or a face thread. It is automatically passed to your UDF by the ANSYS FLUENT solver when you use a `DEFINE` macro that contains a variable thread argument (e.g., `DEFINE_PROFILE`), and the function is hooked to the mixture. Otherwise, if the mixture thread pointer is not explicitly passed to your UDF, then you will need to use another method to retrieve it. For example you can use the `Lookup_Thread` utility macro (see Section 3.2.6: Thread Pointer for Zone ID (`Lookup_Thread`)).

`pt[i]`, an element in the array, is a pointer to the corresponding phase-level thread for the ith phase, where `i` is the `phase_domain_index`. You can use `pt[i]` as an argument to some cell variable macros when you want to retrieve specific phase information at a cell. For example, `C_R(c,pt[i])` can be used to return the density of the ith phase fluid at cell `c`. The pointer `pt[i]` can also be retrieved using `THREAD_SUB_THREAD`, discussed in Section 3.3.2: Phase-Level Thread Pointer (`THREAD_SUB_THREAD`), using i as an argument. The `phase_domain_index` can be retrieved using the `PHASE_DOMAIN_INDEX` macro. See Section 3.3.2: Phase Domain Index (`PHASE_DOMAIN_INDEX`) for details.

Mixture Domain Pointer (`DOMAIN_SUPER_DOMAIN`)

You can use `DOMAIN_SUPER_DOMAIN` when your UDF has access to a particular phase-level domain (subdomain) pointer, and you want to retrieve the mixture-level domain pointer. `DOMAIN_SUPER_DOMAIN` has one argument, `subdomain`. Note that `DOMAIN_SUPER_DOMAIN` is similar in implementation to the `THREAD_SUPER_THREAD` macro described in Section 3.3.2: Mixture Thread Pointer (`THREAD_SUPER_THREAD`).

```
Domain *subdomain;
Domain *mixture_domain = DOMAIN_SUPER_DOMAIN(subdomain);
```

`subdomain` is a pointer to a phase-level domain within the multiphase mixture. It is automatically passed to your UDF by the ANSYS FLUENT solver when you use a `DEFINE` macro that contains a domain variable argument (e.g., `DEFINE_ADJUST`), and the function is hooked to a primary or secondary phase in the mixture. Note that in the current version of ANSYS FLUENT, `DOMAIN_SUPER_DOMAIN` will return the same pointer as

Get_Domain(1). Therefore, if a subdomain pointer is available in your UDF, it is recommended that the DOMAIN_SUPER_DOMAIN macro be used instead of the Get_Domain macro to avoid potential incompatibility issues with future releases of ANSYS FLUENT.

Mixture Thread Pointer (THREAD_SUPER_THREAD)

You can use the THREAD_SUPER_THREAD macro when your UDF has access to a particular phase-level thread (subthread) pointer, and you want to retrieve the mixture-level thread pointer. THREAD_SUPER_THREAD has one argument, subthread.

```
Thread *subthread;
Thread *mixture_thread = THREAD_SUPER_THREAD(subthread);
```

subthread is a pointer to a particular phase-level thread within the multiphase mixture. It is automatically passed to your UDF by the ANSYS FLUENT solver when you use a DEFINE macro that contains a thread variable argument (e.g., DEFINE_PROFILE, and the function is hooked to a primary or secondary phase in the mixture. Note that THREAD_SUPER_THREAD is similar in implementation to the DOMAIN_SUPER_DOMAIN macro described in Section 3.3.2: Mixture Domain Pointer (DOMAIN_SUPER_DOMAIN).

Domain ID (DOMAIN_ID)

You can use DOMAIN_ID when you want to access the domain_id that corresponds to a given phase-level domain pointer. DOMAIN_ID has one argument, subdomain, which is the pointer to a phase-level domain. The default domain_id value for the top-level domain (mixture) is 1. That is, if the domain pointer that is passed to DOMAIN_ID is the mixture-level domain pointer, then the function will return a value of 1. Note that the domain_id that is returned by the macro is the same integer ID that is displayed in the graphical user interface when you select the desired phase in the Phases task page in ANSYS FLUENT.

```
Domain *subdomain;
int domain_id = DOMAIN_ID(subdomain);
```

Phase Domain Index (PHASE_DOMAIN_INDEX)

The PHASE_DOMAIN_INDEX macro retrieves the phase_domain_index for a given phase-level domain (subdomain) pointer. PHASE_DOMAIN_INDEX has one argument, subdomain, which is the pointer to a phase-level domain. phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase.

```
Domain *subdomain;
int phase_domain_index = PHASE_DOMAIN_INDEX(subdomain);
```

3.4   Vector and Dimension Macros

ANSYS FLUENT provides some utilities that you can use in your UDFs to access or manipulate vector quantities and deal with two and three dimensions. These utilities are implemented as macros in the code.

There is a naming convention for vector utility macros. V denotes a vector, S denotes a scalar, and D denotes a sequence of three vector components of which the third is always ignored for a two-dimensional calculation. The standard order of operations convention of parentheses, exponents, multiplication, division, addition, and subtraction (PEMDAS) is not followed in vector functions. Instead, the underscore (_) sign is used to group operands into pairs, so that operations are performed on the elements of pairs before they are performed on groups.

> **i** Note that all of the vector utilities in this section have been designed to work correctly in 2D and 3D. Consequently, you don't need to do any testing to determine this in your UDF.

3.4.1   Macros for Dealing with Two and Three Dimensions

There are two ways that you can deal with expressions involving two and three dimensions in your UDF. The first is to use an explicit method to direct the compiler to compile separate sections of the code for 2D and 3D, respectively. This is done using RP_2D and RP_3D in conditional-if statements. The second method allows you to include general 3D expressions in your UDF, and use ND and NV macros that will remove the z-components when compiling with RP_2D. NV macros operate on vectors while ND macros operate on separate components.

RP_2D **and** RP_3D

The use of a RP_2D and RP_3D macro in a conditional-if statement will direct the compiler to compile separate sections of the code for 2D and 3D, respectively. For example, if you want to direct the compiler to compute swirl terms for the 3D version of ANSYS FLUENT only, then you would use the following conditional compile statement in your UDF:

```
#if RP_3D
   /* compute swirl terms */
#endif
```

3.4.2   The ND **Macros**

The use of ND macros in a UDF allows you to include general 3D expressions in your code, and the ND macros take care of removing the z components of a vector when you are compiling with RP_2D.

ND_ND

The constant ND_ND is defined as 2 for RP_2D (ANSYS FLUENT 2D) and RP_3D (ANSYS FLUENT 3D). It can be used when you want to build a 2×2 matrix in 2D and a 3×3 matrix in 3D. When you use ND_ND, your UDF will work for both 2D and 3D cases, without requiring any modifications.

```
real A[ND_ND][ND_ND]

for (i=0; i<ND_ND; ++i)
  for (j=0; j<ND_ND; ++j)
    A[i][j] = f(i, j);
```

ND_SUM

The utility ND_SUM computes the sum of ND_ND arguments.

```
ND_SUM(x, y, z)

    2D:  x + y;
    3D:  x + y + z;
```

ND_SET

The utility ND_SET generates ND_ND assignment statements.

```
ND_SET(u, v, w, C_U(c, t), C_V(c, t), C_W(c, t))

    u = C_U(c, t);
    v = C_V(c, t);

if 3D:
21
    w = C_W(c, t);
```

3.4.3 The `NV` Macros

The `NV` macros have the same purpose as `ND` macros, but they operate on vectors (i.e., arrays of length `ND_ND`) instead of separate components.

`NV_V`

The utility `NV_V` performs an operation on two vectors.

```
NV_V(a, =, x);

    a[0] = x[0]; a[1] = x[1]; etc.
```

Note that if you use `+ =` instead of `=` in the above equation, then you get

```
    a[0]+=x[0];  etc.
```

See Section 2.6.4: `DEFINE_GRID_MOTION` for an example UDF that utilizes `NV_V`.

`NV_VV`

The utility `NV_VV` performs operations on vector elements. The operation that is performed on the elements depends upon what symbol (`-`,`/`,`*`) is used as an argument in place of the `+` signs in the following macro call.

```
NV_VV(a, =, x, +, y)

    2D:  a[0] = x[0] + y[0], a[1] = x[1] + y[1];
```

See Section 2.6.4: `DEFINE_GRID_MOTION` for an example UDF that utilizes `NV_VV`.

`NV_V_VS`

The utility `NV_V_VS` adds a vector to another vector which is multiplied by a scalar.

```
NV_V_VS(a, =, x, +, y, *, 0.5);

    2D:  a[0] = x[0] + (y[0]*0.5), a[1] = x[1] +(y[1]*0.5);
```

Note that the `+` sign can be replaced by `-`, `/`, or `*`, and the `*` sign can be replaced by `/`.

```
NV_VS_VS
```

The utility `NV_VS_VS` adds a vector to another vector which are each multiplied by a scalar.

```
NV_VS_VS(a, =, x, *, 2.0, +, y, *, 0.5);

    2D:  a[0] = (x[0]*2.0) + (y[0]*0.5), a[1] = (x[1]*2.0) + (y[1]*0.5);
```

Note that the `+` sign can be used in place of `-`, `*`, or `/`, and the `*` sign can be replaced by `/`.

3.4.4   Vector Operation Macros

There are macros that you can use in your UDFs that will allow you to perform operations such as computing the vector magnitude, dot product, and cross product. For example, you can use the `real` function `NV_MAG(V)` to compute the magnitude of vector `V`. Alternatively, you can use the `real` function `NV_MAG2(V)` to obtain the square of the magnitude of vector `V`.

Vector Magnitude Using `NV_MAG` and `NV_MAG2`

The utility `NV_MAG` computes the magnitude of a vector. This is taken as the square root of the sum of the squares of the vector components.

```
NV_MAG(x)

    2D:  sqrt(x[0]*x[0] + x[1]*x[1]);
    3D:  sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```

The utility `NV_MAG2` computes the sum of squares of vector components.

```
NV_MAG2(x)

    2D:  (x[0]*x[0] + x[1]*x[1]);
    3D:  (x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```

See Section 2.5.1: `DEFINE_DPM_BC` for an example UDF that utilizes `NV_MAG`.

Dot Product

The following utilities compute the dot product of two sets of vector components.

```
ND_DOT(x, y, z, u, v, w)

    2D:  (x*u + y*v);
    3D:  (x*u + y*v + z*w);
```

```
NV_DOT(x, u)

    2D:  (x[0]*u[0] + x[1]*u[1]);
    3D:  (x[0]*u[0] + x[1]*u[1] + x[2]*u[2]);
```

```
NVD_DOT(x, u, v, w)

    2D:  (x[0]*u + x[1]*v);
    3D:  (x[0]*u + x[1]*v + x[2]*w);
```

See Section 2.3.6: DEFINE_DOM_SPECULAR_REFLECTIVITY for an example UDF that utilizes NV_DOT.

Cross Product

For 3D, the CROSS macros return the specified component of the vector cross product. For 2D, the macros return the cross product of the vectors with the z-component of each vector set to 0.

```
ND_CROSS_X(x0,x1,x2,y0,y1,y2)
    2D:  0.0
    3D:  (((x1)*(y2))-(y1)*(x2)))
```

```
ND_CROSS_Y(x0,x1,x2,y0,y1,y2)
    2D:  0.0
    3D:  (((x2)*(y0))-(y2)*(x0)))
```

```
ND_CROSS_Z(x0,x1,x2,y0,y1,y2)
    2D and 3D: (((x0)*(y1))-(y0)*(x1)))
```

```
NV_CROSS_X(x,y)
```

```
    ND_CROSS_X(x[0],x[1],x[2],u[0],y[1],y[2])

NV_CROSS_Y(x,y)
    ND_CROSS_X(x[0],x[1],x[2],u[0],y[1],y[2])

NV_CROSS_Z(x,y)
    ND_CROSS_X(x[0],x[1],x[2],u[0],y[1],y[2])

NV_CROSS(a,x,y)

    a[0] = NV_CROSS_X(x,y);
    a[1] = NV_CROSS_Y(x,y);
    a[2] = NV_CROSS_Z(x,y);
```

See Section 2.6.4: DEFINE GRID MOTION for an example UDF that utilizes NV CROSS.

3.5 Time-Dependent Macros

You can access time-dependent variables in your UDF in two different ways: direct access using a solver macro, or indirect access using an RP variable macro. Table 3.5.1 contains a list of solver macros that you can use to access time-dependent variables in ANSYS FLUENT. An example of a UDF that uses a solver macro to access a time-dependent variable is provided below. See Section 2.2.2: DEFINE DELTAT for another example that utilizes a time-dependent macro.

Table 3.5.1: Solver Macros for Time-Dependent Variables

Macro Name	Returns
CURRENT_TIME	real current flow time (in seconds)
CURRENT_TIMESTEP	real current physical time step size (in seconds)
PREVIOUS_TIME	real previous flow time (in seconds)
PREVIOUS_2_TIME	real flow time two steps back in time (in seconds)
PREVIOUS_TIMESTEP	real previous physical time step size (in seconds)
N_TIME	integer number of time steps
N_ITER	integer number of iterations

i You *must* include the unsteady.h header file in your UDF source code when using the PREVIOUS TIME or PREVIOUS 2 TIME macros since it is not included in udf.h.

i N_ITER can only be utilized in compiled UDFs.

Some time-dependent variables such as current physical flow time can be accessed directly using a solver macro (CURRENT_TIME), or indirectly by means of the RP variable macro RP_Get_Real("flow-time"). These two methods are shown below.

Solver Macro Usage

```
real current_time;
current_time = CURRENT_TIME;
```

"Equivalent" RP Macro Usage

```
real current_time;
current_time = RP_Get_Real("flow-time");
```

Table 3.5.2 shows the correspondence between solver and RP macros that access the same time-dependent variables.

Table 3.5.2: Solver and RP Macros that Access the Same Time-Dependent Variable

Solver Macro	"Equivalent" RP Variable Macro
CURRENT_TIME	RP_Get_Real("flow-time")
CURRENT_TIMESTEP	RP_Get_Real("physical-time-step")
N_TIME	RP_Get_Integer("time-step")

i You should not access a scheme variable using any of the RP_GET_... functions from inside a cell or face looping macro (c_loop or f_loop). This type of communication between the solver and cortex is very time consuming and therefore should be done outside of loops.

Example

The integer time step count (accessed using N_TIME) is useful in DEFINE_ADJUST functions for detecting whether the current iteration is the first in the time step.

```
/***********************************************************************
            Example UDF that uses N_TIME
***********************************************************************/
static int last_ts = -1;   /*  Global variable.  Time step is never <0 */

DEFINE_ADJUST(first_iter_only, domain)
{
  int curr_ts;
  curr_ts = N_TIME;
  if (last_ts != curr_ts)
  {
    last_ts = curr_ts;

    /* things to be done only on first iteration of each time step
       can be put here  */
  }
}
```

> **i** There is a new variable named first_iteration that can be used in the above if statement. first_iteration is true only at the first iteration of a timestep. Since the adjust UDF is also called before timestepping begins, the two methods vary slightly as to when they are true. You must decide which behavior is more appropriate for your case.

3.6 Scheme Macros

The text interface of ANSYS FLUENT executes a Scheme interpreter which allows you to define your own variables that can be stored in ANSYS FLUENT and accessed via a UDF. This capability can be very useful, for example, if you want to alter certain parameters in your case, and you do not want to recompile your UDF each time. Suppose you want to apply a UDF to multiple zones in a mesh. You can do this manually by accessing a particular Zone ID in the graphical user interface, hardcoding the integer ID in your UDF, and then recompiling the UDF. This can be a tedious process if you want to apply the UDF to a number of zones. By defining your own scheme variable, if you want to alter the variable later, then you can do it from the text interface using a Scheme command.

Macros that are used to define and access user-specified Scheme variables from the text interface are identified by the prefix `rp`, (e.g., `rp-var-define`). Macros that are used to access user-defined Scheme variables in an ANSYS FLUENT solver, are identified by the prefix `RP` (e.g., `RP_Get_Real`). These macros are executed within UDFs.

3.6.1 Defining a Scheme Variable in the Text Interface

To define a scheme variable named `pres_av/thread-id` in the text interface, you can use the scheme command:

```
(rp-var-define 'pres_av/thread-id 2 'integer #f)
```

Before you define a scheme variable, it is often good practice to check that the variable is not already defined. You can do this by typing the following command in the text window:

```
(if (not (rp-var-object 'pres_av/thread-id))
    (rp-var-define 'pres_av/thread-id 2 'integer #f))
```

This command first checks that the variable `pres_av/thread-id` is not already defined, and then sets it up as an integer with an initial value of 2.

Note that the string `'/'` is allowed in Scheme variable names (as in `pres_av/thread-id`), and is a useful way to organize variables so that they do not interfere with each other.

3.6.2   Accessing a Scheme Variable in the Text Interface

After you define a Scheme variable in the text interface, you can access the variable. For example, if you want to check the current value of the variable (e.g., `pres_av/thread-id`) on the Scheme side, you can type the following command in the text window:

```
(%rpgetvar 'pres_av/thread-id)
```

i It is recommended that you use %rpgetvar when you are retrieving an ANSYS FLUENT variable using a scheme command. This will ensure that you access the current cached value.

3.6.3   Changing a Scheme Variable to Another Value in the Text Interface

Alternatively, if you want to change the value of the variable you have defined (`pres_av/thread-id`) to say, 7, then you will need to use `rpsetvar` and issue the following command in the text window:

```
(rpsetvar 'pres_av/thread-id 7)
```

3.6.4   Accessing a Scheme Variable in a UDF

After a new variable is defined on the Scheme side (using a text command), you will need to bring it over to the solver side to be able to use it in your UDF. 'RP' macros are used to access Scheme variables in UDFs, and are listed below.

`RP_Get_Real("variable-name")`	Returns the double value of `variable-name`
`RP_Get_Integer("variable-name")`	Returns the integer value of `variable-name`
`RP_Get_String("variable-name")`	Returns the char* value of `variable-name`
`RP_Get_Boolean("variable-name")`	Returns the Boolean value of `variable-name`

For example, to access the user-defined Scheme variable `pres_av/thread-id` in your UDF C function, you will use `RP_Get_Integer`. You can then assign the variable returned to a local variable you have declared in your UDF (e.g., `surface_thread_id`) as demonstrated below:

```
surface_thread_id = RP_Get_Integer("pres_av/thread-id");
```

3.7   Input/Output Macros

ANSYS FLUENT provides some utilities in addition to the standard C I/O functions that you can use to perform input/output (I/O) tasks. These are listed below and are described in the following sections:

```
Message(format, ...)   prints a message to the console
Error(format, ...)     prints an error message to the console
```

Message

The `Message` macro is a utility that displays data to the console in a format that you specify.

```
int Message(char *format, ...);
```

The first argument in the `Message` macro is the format string. It specifies how the remaining arguments are to be displayed in the console. The format string is defined within quotes. The value of the replacement variables that follow the format string will be substituted in the display for all instances of `%type`. The `%` character is used to designate the character type. Some common format characters are: `%d` for integers, `%f` for floating point numbers, `%g` for double data type, and `%e` for floating point numbers in exponential format (with `e` before the exponent). Consult a C programming language manual for more details. The format string for `Message` is similar to `printf`, the standard C I/O function (see Section A.13.3: Standard I/O Functions for details).

In the example below, the text `Volume integral of turbulent dissipation:` will be displayed in the console, and the value of the replacement variable, `sum_diss`, will be substituted in the message for all instances of `%g`.

Example:

```
Message("Volume integral of turbulent dissipation: %g\n", sum_diss);
    /*  g represents floating point number in f or e format */
    /*  \n denotes a new line  */
```

> *i* It is recommended that you use `Message` instead of `printf` in compiled UDFs (UNIX only).

Error

You can use `Error` when you want to stop execution of a UDF and print an error message to the console.

Example:

```
if (table_file == NULL)
   Error("error reading file");
```

> $\boxed{i}$ `Error` is not supported by the interpreter and can be used only in compiled UDFs.

3.8 Miscellaneous Macros

N_UDS

You can use `N_UDS` to access the number of user-defined scalar (UDS) transport equations that have been specified in ANSYS FLUENT. The macro takes no arguments and returns the integer number of equations. It is defined in `models.h`.

N_UDM

You can use `N_UDM` to access the number of user-defined memory (UDM) locations that have been used in ANSYS FLUENT. The macro takes no arguments, and returns the integer number of memory locations used. It is defined in `models.h`.

Data_Valid_P()

You can check that the cell values of the variables that appear in your UDF are accessible before you use them in a computation by using the `Data_Valid_P` macro.

```
cxboolean Data_Valid_P()
```

`Data_Valid_P` is defined in the `id.h` header file, and is included in `udf.h`. The function returns `1` (true) if the data that is passed as an argument is valid, and `0` (false) if it is not.

Example:

```
if(!Data_Valid_P())  return;
```

Suppose you read a case file and, in the process, load a UDF. If the UDF performs a calculation using variables that have not yet been initialized, such as the velocity at interior cells, then an error will occur. To avoid this kind of error, an `if else` condition can be added to your code. If (`if`) the data are available, the function can be computed in the normal way. If the data are not available (`else`), then no calculation, or a trivial calculation can be performed instead. After the flow field has been initialized, the function can be reinvoked so that the correct calculation can be performed.

FLUID_THREAD_P()

```
cxboolean FLUID_THREAD_P(t);
```

You can use `FLUID_THREAD_P` to check whether a cell thread is a fluid thread. The macro is passed a cell thread pointer `t`, and returns `1` (or `TRUE`) if the thread that is passed is a fluid thread, and `0` (or `FALSE`) if it is not.

Note that `FLUID_THREAD_P(t)` assumes that the thread is a cell thread.

For example,

```
FLUID_THREAD_P(t0);
```

returns `TRUE` if the thread pointer `t0` passed as an argument represents a fluid thread.

NULLP & NNULLP

You can use the `NULLP` and `NNULLP` functions to check whether storage has been allocated for user-defined scalars. `NULLP` returns `TRUE` if storage is *not* allocated, and `NNULLP` returns `TRUE` if storage is allocated. Below are some examples of usage.

```
NULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))

/*  NULLP returns TRUE if storage is not allocated for
    user-defined storage variable                      */


NNULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))

/*  NNULLP returns TRUE if storage is allocated for
    user-defined storage variable                      */
```

`M_PI`

The macro `M_PI` returns the value of π.

`UNIVERSAL_GAS_CONSTANT`

`UNIVERSAL_GAS_CONSTANT` returns the value of the universal gas constant (8314.34 $J/Kmol$-K).

> *i*   Note that this constant is *not* expressed in SI units.

See Section 2.3.27: `DEFINE_VR_RATE` for an example UDF that utilizes `UNIVERSAL_GAS_CONSTANT`.

`SQR(k)`

`SQR(k)` returns the square of the given variable `k`, or $k * k$.

Chapter 4. Interpreting UDFs

After you have written your UDF using any text editor and have saved the source code file it with a `.c` extension in your working folder, you are ready to interpret the source file. Follow the instructions below in Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box. After it has been interpreted, the UDF function name(s) that you supplied in the `DEFINE` macro(s) will appear in drop-down lists in graphical dialog boxes in ANSYS FLUENT, ready for you to hook to your CFD model. Alternatively, if you wish to compile your UDF source file, see Chapter 5: Compiling UDFs for details.

- Section 4.1: Introduction

- Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box

- Section 4.3: Common Errors Made While Interpreting A Source File

4.1 Introduction

An interpreted UDF is a function that is interpreted directly from a source file (e.g., `udfexample.c`) at *runtime*. You will use the Interpreted UDFs dialog box to interpret all of the functions in a source file (e.g., udfexample.c) in a single step. After a source file is interpreted, you can write the case file and the names and contents of the interpreted function(s) will be stored in the case. In this way, the function(s) will be *automatically* interpreted whenever the case file is subsequently read. After it has been interpreted (either manually through the Interpreted UDFs dialog box or automatically upon reading a case file), all of the interpreted UDFs that are contained within a source file will become visible and selectable in graphical user interface dialog boxes in ANSYS FLUENT.

Inside ANSYS FLUENT, the source code is compiled into an intermediate, architecture-independent machine code using a C preprocessor. This machine code then executes on an internal emulator, or interpreter, when the UDF is invoked. This extra layer of code incurs a performance penalty, but allows an interpreted UDF to be shared effortlessly between different architectures, operating systems, and ANSYS FLUENT versions. If execution speed does become an issue, an interpreted UDF can always be run in compiled mode without modification.

4.1.1 Location of the `udf.h` File

UDFs are defined using `DEFINE` macros (see Chapter 2: DEFINE Macros) and the definitions for `DEFINE` macros are included in `udf.h` header file. Consequently, before you can interpret a UDF source file, `udf.h` will need to be accessible in your path, or saved locally within your working folder.

The location of the `udf.h` file is:

$$path\verb|\ANSYS Inc\v120\fluent\fluent12.0.|\overset{\Downarrow}{x}\verb| \src\udf.h|$$

where *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is `C:\Program Files`), and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

> **i** In general, you should not copy `udf.h` from the installation area. The compiler is designed to look for this file locally (in your current folder) first. If it is not found in your current folder, the compiler will look in the `\src` folder automatically. In the event that you upgrade your release area, but do not remove an old copy of `udf.h` from your working folder, you will not be accessing the most recent version of this file.

> **i** You should not, under any circumstances, alter the `udf.h` file.

4.1.2 Limitations

Due to limitations in the interpreter used to compile interpreted UDF source code in ANSYS FLUENT, interpreted UDFs are limited in their use of the C programming language. In particular, the following elements of C cannot be used in interpreted UDFs:

- goto statements
- non ANSI-C prototypes for syntax
- direct data structure references
- declarations of local structures
- unions
- pointers to functions
- arrays of functions
- multi-dimensional arrays

4.2 Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box

This section presents the steps for interpreting a source file in ANSYS FLUENT. After it has been interpreted, the names of UDFs contained within the source file will appear in drop-down lists in graphics dialog boxes in ANSYS FLUENT.

The general procedure for interpreting a source file is as follows:

1. Make sure that the UDF source file is in the same folder that contains your case and data files.

   i If you are running the parallel version of ANSYS FLUENT on a network of Windows machines, you must 'share' the working folder that contains your UDF source, case, and data files so that all of the compute nodes in the cluster can see it. To share the working folder, open Windows Explorer and browse to the folder; right-click on the working folder, select Sharing and Security from the menu, click Share this folder, and click OK.

2. For UNIX/Linux, start ANSYS FLUENT from the directory that contains your case, data, and UDF source files. For Windows, start ANSYS FLUENT using FLUENT Launcher, being sure to specify the folder that contains your case, data, and UDF source files in the Working Directory text box in the General Options tab.

3. Read (or set up) your case file.

4. Interpret the UDF using the Interpreted UDFs dialog box (Figure 4.2.1).

   Define ⟶ User-Defined ⟶ Functions ⟶ Interpreted...
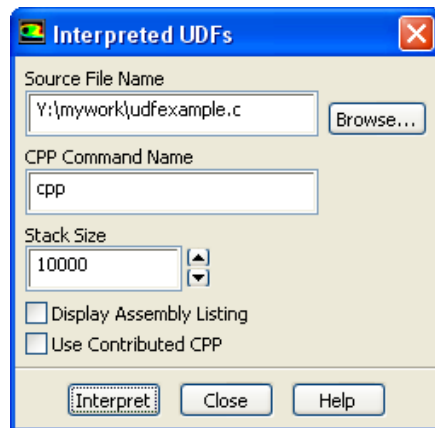
Figure 4.2.1: The Interpreted UDFs Dialog Box

(a) Indicate the UDF source file you want to interpret by clicking the Browse...
button. This will open the Select File dialog box (Figure 4.2.2).
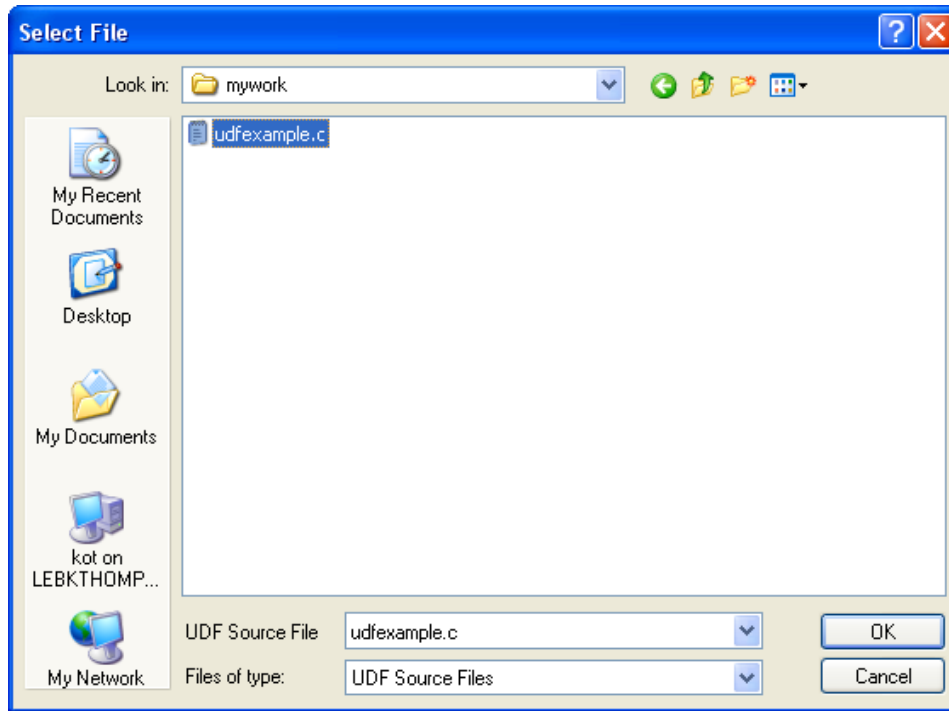
Figure 4.2.2: The Select File Dialog Box

In the Select File dialog box, select the desired file (e.g., udfexample.c) and
click OK. The Select File dialog box will close and the complete path to the
file you selected will appear in the Source File Name text box in the Interpreted
UDFs dialog box (Figure 4.2.1).

(b) In the Interpreted UDFs dialog box, specify the C preprocessor to be used in
the CPP Command Name text box. You can keep the default cpp or you can
enable the Use Contributed CPP option to use the preprocessor supplied by
ANSYS FLUENT.

(c) Keep the default Stack Size setting of 10000, unless the number of local vari-
ables in your function will cause the stack to overflow. In this case, set the
Stack Size to a number that is greater than the number of local variables used.

(d) Enable the Display Assembly Listing option on if you want a listing of assembly
language code to appear in the console when the function interprets. This
option will be saved in your case file, so that when you read the case in a
subsequent ANSYS FLUENT session, the assembly code will be automatically
displayed.

(e) Click Interpret to interpret your UDF.

If the compilation is successful and you have enabled Display Assembly Listing, then the assembler code will be displayed in the console. If you chose not to display the listing and the compilation is successful, then the CPP Command Name that was executed will be displayed the console. If the compilation is unsuccessful, then ANSYS FLUENT will report an error and you will need to debug your program. See Section 4.3: Common Errors Made While Interpreting A Source File. You can also view the compilation history in the log file that is saved in your working folder.

(f) Close the Interpreted UDFs Dialog Box when the interpreter has finished.

5. Write the case file. The interpreted function(s) will be saved with the case file, and *automatically* interpreted when the case file is subsequently read.

4.3 Common Errors Made While Interpreting A Source File

If there are compilation errors when you interpret a UDF source file, they will appear in the console. However, you may not see all the error messages if they scroll off the screen too quickly. For this reason, you may want to disable the Display Assembly Listing option while debugging your UDF. You can view the compilation history in the log file that is saved in your working folder.

If you keep the Interpreted UDFs dialog box open while you are in the process of debugging your UDF, the Interpret button can be used repeatedly since you can make changes with an editor in a separate window. Then, you can continue to debug and interpret until no errors are reported. Remember to save changes to the source code file in the editor window before trying to interpret again.

One of the more common errors made when interpreting source files is trying to interpret code that contains elements of C that the interpreter does not accommodate. For example, if you have code that contains a structured reference call (which is not supported by the C preprocessor), the interpretation will fail and you will get an error message similar to the following:

```
Error: /nfs/clblnx/home/clb/fluent/udfexample.c:
line 15: structure reference
```

Chapter 5.   Compiling UDFs

After you have written your UDF(s) using any text editor and have saved the source file with a `.c` extension in your working folder, you are ready to compile the UDF source file, build a shared library from the resulting objects, and load the library into ANSYS FLUENT. After being loaded, the function(s) contained in the library will appear in drop-down lists in graphical interface dialog boxes, ready for you to hook to your CFD model. Follow the instructions in Section 5.2: Compiling a UDF Using the GUI to compile UDF source files using the graphical user interface (GUI). Section 5.3: Compile a UDF Using the TUI explains how you can use the text user interface (TUI) to do the same. The text interface option provides the added capability of allowing you to link precompiled object files derived from non-ANSYS FLUENT sources (e.g., Fortran sources) to your UDF (Section 5.4: Link Precompiled Object Files From Non-ANSYS FLUENT Sources). This feature is not available in the GUI. Section 5.5: Load and Unload Libraries Using the UDF Library Manager Dialog Box describes how you can load (and unload) multiple UDF libraries using the UDF Library Manager dialog box. The capability of loading more than one UDF library into ANSYS FLUENT raises the possibility of data contention if multiple libraries use the same user-defined scalar (UDS) and user-defined memory (UDM) locations. These clashes can be avoided if libraries reserve UDS or UDM prior to usage. See Sections 3.2.8 and 3.2.9, respectively, for details.

- Section 5.1: Introduction

- Section 5.2: Compiling a UDF Using the GUI

- Section 5.3: Compile a UDF Using the TUI

- Section 5.4: Link Precompiled Object Files From Non-ANSYS FLUENT Sources

- Section 5.5: Load and Unload Libraries Using the UDF Library Manager Dialog Box

- Section 5.6: Common Errors When Building and Loading a UDF Library

- Section 5.7: Special Considerations for Parallel ANSYS FLUENT

5.1 Introduction

Compiled UDFs are built in the same way that the ANSYS FLUENT executable itself is built. Internally, a script called `Makefile` is used to invoke the system C compiler to build an object code library that contains the native machine language translation of your higher-level C source code. The object library is specific to the computer architecture being used during the ANSYS FLUENT session, as well as to the particular version of the ANSYS FLUENT executable being run. Therefore, UDF object libraries must be rebuilt any time ANSYS FLUENT is upgraded, when the computer's operating system level changes, or when the job is run on a different type of computer architecture. The generic process for compiling a UDF involves two steps: compile/build and load.

The compile/build step takes one or more source files (e.g., `myudf.c`) containing at least one UDF and compiles them into object files (e.g., `myudf.o` or `myudf.obj`) and then builds a "shared library" (e.g., `libudf`) with the object files. If you compile your source file using the GUI, this compile/build process is executed when you click the Build button in the Compiled UDFs dialog box. The shared library that you name (e.g., `libudf`) is automatically built for the architecture and version of ANSYS FLUENT you are running during that session (e.g., `hpux11/2d`), and will store the UDF object file(s).

If you compile your source file using the TUI, you will first need to setup target folders for the shared libraries, modify a file named `makefile` to specify source parameters, and then execute the `Makefile` which directs the compile/build process. Compiling a UDF using the TUI has the added advantage of allowing precompiled object files derived from non-ANSYS FLUENT sources to be linked to ANSYS FLUENT (Section 5.4: Link Precompiled Object Files From Non-ANSYS FLUENT Sources). This option is not available using the GUI.

After the shared library is built (using the TUI or GUI) you will need to load the UDF library into ANSYS FLUENT before you can use it. This can be done using the Load button in the Compiled UDFs dialog box. After being loaded, all of the compiled UDFs that are contained within the shared library will become visible and selectable in graphics dialog boxes in ANSYS FLUENT. Note that compiled UDFs are displayed in ANSYS FLUENT dialog boxes with the associated UDF library name separated by two colons (`::`). For example, a compiled UDF named `rrate` that is associated with a shared library named `libudf` would appear in ANSYS FLUENT dialog boxes as `rrate::libudf`. This distinguishes UDFs that are compiled from those that are interpreted.

If you write your case file when a UDF library is loaded, the library will be saved with the case and will be *automatically* loaded whenever that case file is subsequently read. This process of "dynamic loading" saves you having to reload the compiled library every time you want to run a simulation.

Before you compile your UDF source file(s) using one of the two methods provided in Sections 5.2 and 5.3, you will first need to make sure that the `udf.h` header file is accessible in your path, or is saved locally within your working folder (Section 5.1.1: Location of the `udf.h` File).

5.1.1   Location of the `udf.h` File

UDFs are defined using `DEFINE` macros (see Chapter 2: `DEFINE` Macros) and the definitions for `DEFINE` macros are included in `udf.h`. Consequently, before you compile your source file, the `udf.h` header file will need to be accessible in your path, or saved locally within your working folder.

The location of the `udf.h` file is:

$$path\backslash\texttt{ANSYS Inc}\backslash\texttt{v120}\backslash\texttt{fluent}\backslash\texttt{fluent12.0.}\overset{\Downarrow}{x}\backslash\texttt{src}\backslash\texttt{udf.h}$$

where *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is `C:\Program Files`), and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

> **i** In general, you should not copy `udf.h` from the installation area. The compiler is designed to look for this file locally (in your current folder) first. If it is not found in your current folder, the compiler will look in the `\src` folder automatically. In the event that you upgrade your release area, but do not remove an old copy of `udf.h` from your working folder, you will not be accessing the most recent version of this file.

> **i** You should not, under any circumstances, alter the `udf.h` file.

There may be instances when will want to include additional header files in the compilation process. Make sure that all header files needed for UDFs are located in the `\src` folder.

5.1.2 Compilers

The graphical and text interface processes for a compiled UDF require the use of a C compiler that is native to the operating system and machine you are running on. Most UNIX operating systems provide a C compiler as a standard feature. If you are operating on a Windows system, you will need to ensure that a Microsoft Visual Studio is installed on your machine before you proceed. If you are unsure about compiler requirements for your system, please contact ANSYS FLUENT installation support. For Linux machines, ANSYS FLUENT supports any ANSI-compliant compiler.

i Obsolete versions of any native compiler may not work properly with compiled UDFs.

When launching ANSYS FLUENT on Windows using FLUENT Launcher, the UDF Compiler tab (Figure 5.1.1) allows you to specify compiler settings for compiling UDFs.
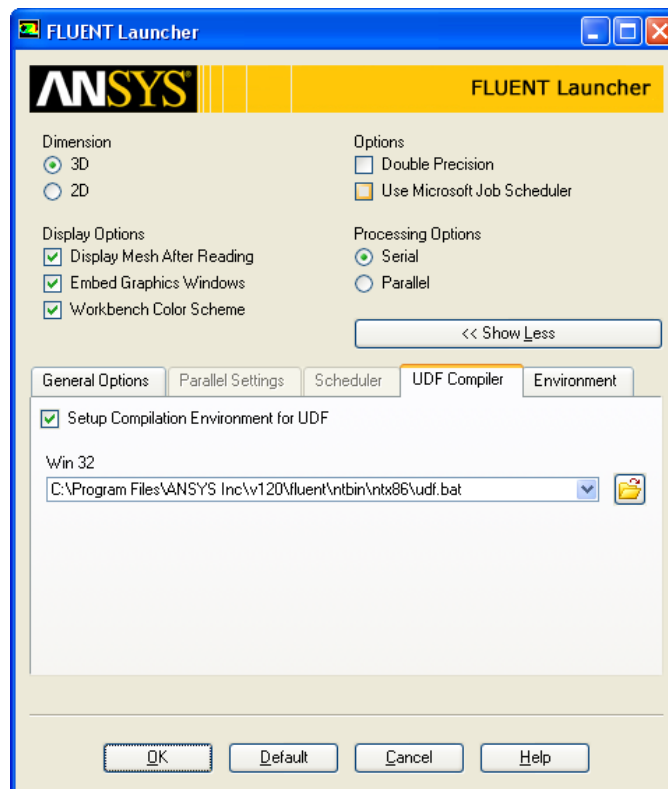
Figure 5.1.1: The UDF Compiler Tab of the FLUENT Launcher Dialog Box

The Setup Compilation Environment for UDF option is enabled by default, and allows you to specify a batch file that contains UDF compilation environment settings. Enter a batch file name and path in the Win 32 text box, or click [icon] to browse for a batch file.

By default, the FLUENT Launcher dialog box is set to use `udf.bat` file that is that is saved in your computer as part of the ANSYS FLUENT installation. It is recommended that you keep the default batch file, which is tested with the latest MS Visual Studio C++ compilers at the time of the ANSYS FLUENT release date.

5.2 Compiling a UDF Using the GUI

The general procedure for compiling a UDF source file and building a shared library for the resulting objects, and loading the compiled UDF library into ANSYS FLUENT using the graphical user interface (GUI) is as follows.

> i Note that in order to compile a UDF when running serial or parallel AN-SYS FLUENT on a Windows system, then you must have Microsoft Visual Studio installed on your machine, preferably on the `C:` drive.

1. Make sure that the UDF source file you want to compile is in the same folder that contains your case and data files.

   > i Note that if you wish to compile a UDF while running ANSYS FLUENT on a Windows parallel network, then you *must* 'share' the working folder where the UDF is located so that all computers on the cluster can see this folder. To share the working folder, open Windows Explorer and browse to the folder; right-click on the working folder, select Sharing and Security from the menu, click Share this folder, and click OK. If you forget to enable the sharing option for the folder using Windows Explorer, then ANSYS FLUENT will hang when you try to load the library in the Compiled UDFs dialog box.

2. For UNIX/Linux, start ANSYS FLUENT from the directory that contains your case, data, and UDF source files. For Windows, start ANSYS FLUENT using FLUENT Launcher with the following settings:

   • Specify the folder that contains your case, data, and UDF source files in the Working Directory text box in the General Options tab.

   • Make sure that the batch file for the UDF compilation environment settings is correctly specified in the UDF Compiler tab (see Section 5.1.2: Compilers for further details).

3. Read (or set up) your case file.

4. Open the Compiled UDFs dialog box (Figure 5.2.1).

$\boxed{\text{Define}} \longrightarrow \boxed{\text{User-Defined}} \longrightarrow \boxed{\text{Functions}} \longrightarrow \text{Compiled...}$
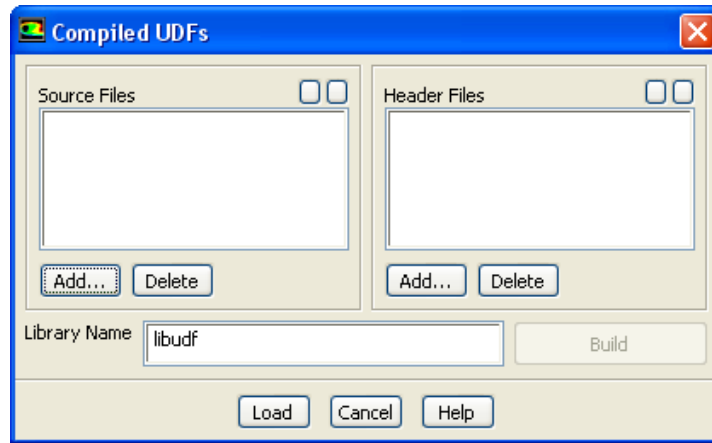
Figure 5.2.1: The Compiled UDFs Dialog Box

5. In the Compiled UDFs dialog box click on Add... under Source Files to select the UDF source file (or files) you want to compile. This will open the Select File dialog box (shown in Figure 5.2.2).

6. In the Select File dialog box, click the names of all of the desired files (e.g., udfex-ample.c), so that the complete paths to the source files are displayed under Source File(s). You can remove a selection by clicking the path in Source File(s) list and then clicking the Remove button. Click OK when your selections are complete.

The Select File dialog box will close and the file you selected (e.g., `udfexample.c`) will appear in the Source Files list in the Compiled UDFs dialog box (Figure 5.2.3). You can delete a file after adding it by selecting the source file and then clicking Delete in the Compiled UDFs dialog box.
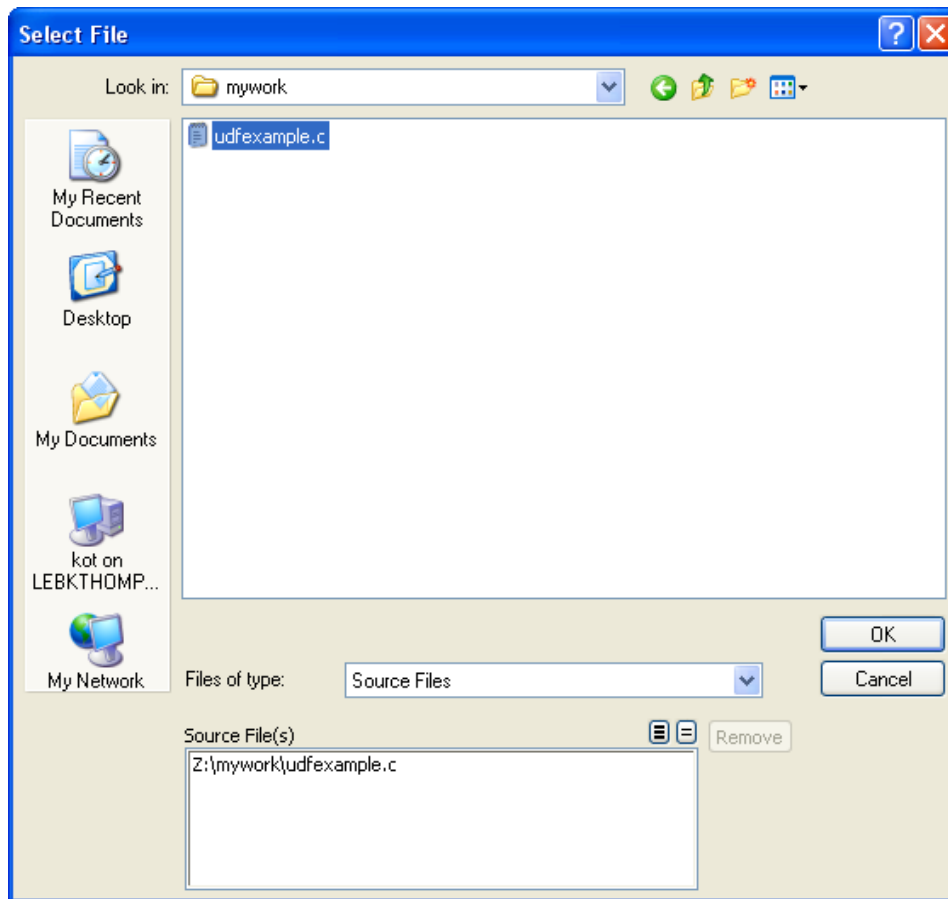
Figure 5.2.2: The Select File Dialog Box
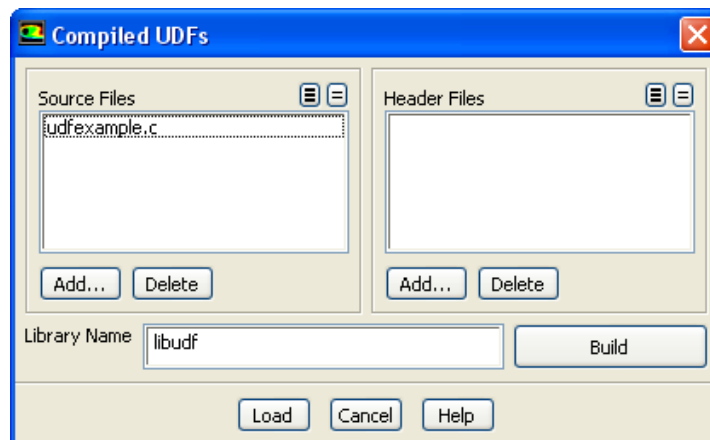
Figure 5.2.3: The Compiled UDFs Dialog Box

7. In the Compiled UDFs dialog box, select additional header files that you want to include in the compilation by clicking Add... under Header File(s) and repeat the previous step.

8. In the Compiled UDFs dialog box (Figure 5.2.3), enter the name of the shared library you want to build in the Library Name field (or leave the default name libudf), and click Build. All of the UDFs that are contained within each C source file you selected will be compiled and the build files will be stored in the shared library you specified (e.g., libudf).

    As the compile/build process begins, a Warning dialog box (Figure 5.2.4) will appear reminding you that the source file(s) need to be in the same folder as the case and data files. Click OK to close the dialog and continue with the build.
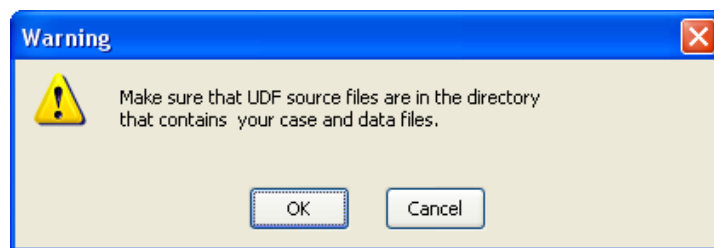
Figure 5.2.4: The Warning Dialog Box

As the build process progresses, the results of the build will be displayed on the console. You can also view the compilation history in the log file that is saved in your working folder.

Console messages for a successful compile/build for a source file named `udfexample.c` and a UDF library named `libudf` for a Windows architecture are shown below.

```
Deleted old libudf\ntx86\2d\libudf.dll
        1 file(s) copied.
(system "copy C:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\src
\makefile_nt.udf
libudf\ntx86\2d\makefile")
        1 file(s) copied.
(chdir "libudf")()
(chdir "ntx86\2d")()
udfexample.c
# Generating udf_names.c because of makefile udfexample.obj
udf_names.c
# Linking libudf.dll because of makefile user_nt.udf udf_names.obj
 udfexample.obj
```

```
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

    Creating library libudf.lib and object libudf.exp

Done.
```

9. In the Compiled UDFs dialog box (Figure 5.2.3), load the shared library that was just built into ANSYS FLUENT by clicking Load.

   A message will be displayed on the console providing a status of the load process. For example:

```
"Z:/mywork"

Opening library "libudf"...
Library "libudf\ntx86\2d\libudf.dll" opened
inlet_x_velocity
Done.
```

   indicates that the shared library named libudf was successfully loaded (on a Windows machine) and it contains one function named inlet_x_velocity.

   > *i* Note that compiled UDFs are displayed in ANSYS FLUENT dialog boxes with the associated UDF library name using the :: identifier. For example, a compiled UDF named inlet_x_velocity that is associated with a shared library named libudf will appear in ANSYS FLUENT dialog boxes as inlet_x_velocity::libudf. This visually distinguishes UDFs that are compiled from those that are interpreted.

   After the compiled UDF(s) become visible and selectable in graphics dialog boxes in ANSYS FLUENT, they can be hooked to your model. See Chapter 6: Hooking UDFs to ANSYS FLUENT for details. You can use the UDF Library Manager dialog box to unload the shared library, if desired. See Section 5.5: Load and Unload Libraries Using the UDF Library Manager Dialog Box for details.

10. Write the case file if you want the compiled function(s) in the shared library to be saved with the case. The functions will be loaded *automatically* into ANSYS FLUENT whenever the case is subsequently read.

   > *i* If you do not want the shared library saved with your case file, then you must remember to load it into ANSYS FLUENT using the Compiled UDFs dialog box or the UDF Library Manager dialog box in subsequent sessions.

5.3 Compile a UDF Using the TUI

The first step in compiling a UDF source file using the text user interface (TUI) involves setting up the folder structure where the shared (compiled) library will reside, for each of the versions of ANSYS FLUENT you wish to run (e.g., 2d, 3d). You will then modify the file named `makefile` to setup source file parameters. Subsequently, you will execute the `Makefile` which compiles the source file and builds the shared library from the resulting object files. Finally, you will load the UDF library into ANSYS FLUENT. Using the TUI option allows you the added advantage of building a shared library for precompiled object file(s) that are derived from non-ANSYS FLUENT sources (e.g., `.o` objects from `.f` sources). See Section 5.4: Link Precompiled Object Files From Non-ANSYS FLUENT Sources for details.

> *i* Note that if you are running serial or parallel ANSYS FLUENT on a Windows system and intend to compile a UDF, then you must have Microsoft Visual Studio installed on your machine, preferably on the `C:` drive.

5.3.1 Set Up the Directory Structure

The folder/directory structures for Windows systems and UNIX/Linux systems are different, so the procedure for setting up the folder/directory structure is described separately for each.

Windows Systems

For compiled UDFs on Windows systems, two ANSYS FLUENT files are required to build your shared UDF library: `makefile_nt.udf` and `user_nt.udf`. The file `user_nt.udf` has a user-modifiable section that allows you to specify source file parameters.

The procedure below outlines steps that you need to follow in order to set up the folder structure required for the shared library.

1. In your working folder, make a folder that will store your UDF library (e.g., `libudf`).

2. Make a folder below this called `src`.

3. Put all your UDF source files into this folder (e.g., `libudf\src`).

4. Make an architecture folder below the library folder called `ntx86` or `win64` for Intel systems running Windows (e.g., `libudf\ntx86`).

5. In the architecture folder (e.g., `libudf\ntx86`), create folders for the ANSYS FLU-ENT versions you want to build for your architecture. (e.g., `ntx86\2d` and `ntx86\3d`). Possible versions are:

`2d` or `3d`	single-precision serial 2D or 3D
`2ddp` or `3ddp`	double-precision serial 2D or 3D
`2d_node` and `2d_host`	single-precision parallel 2D
`3d_node` and `3d_host`	single-precision parallel 3D
`2ddp_node` and `2ddp_host`	double-precision parallel 2D
`3ddp_node` and `3ddp_host`	double-precision parallel 3D

> **i** Note that you must create *two* build folders for each parallel version of the solver (two for the 3D version, two for the 2D double-precision version, etc.), regardless of the number of compute nodes.

6. Copy `user_nt.udf` from

$$path\texttt{\textbackslash ANSYS Inc\textbackslash v120\textbackslash fluent\textbackslash fluent12.0.}\overset{\Downarrow}{x}\texttt{ \textbackslash src\textbackslash}$$

to all the version subfolders you have made (e.g., `libudf\ntx86\3d`).

Note that *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is `C:\Program Files`), and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

7. Copy `makefile_nt.udf` from

$$path\texttt{\textbackslash ANSYS Inc\textbackslash v120\textbackslash fluent\textbackslash fluent12.0.}\overset{\Downarrow}{x}\texttt{ \textbackslash src\textbackslash}$$

to all the version subfolders you have made (e.g., `libudf\ntx86\3d`) and rename it `makefile`.

Note that *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is `C:\Program Files`), and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

UNIX and Linux Systems

For compiled UDFs on UNIX systems, two ANSYS FLUENT files are required to build your shared UDF library: `makefile.udf` and `makefile.udf2`. The file `makefile` has a user-modifiable section that allows you to specify source file parameters. The procedure below outlines steps that you need to follow in order to set up the directory structure required for the shared library.

1. In your working directory, make a directory that will store your UDF library (e.g., `libudf`).

2. Copy `makefile.udf2` from

$$path/\texttt{ansys\_inc/v120/fluent/fluent12.0.}\overset{\Downarrow}{x}\texttt{/src/makefile.udf2}$$

   to the library directory (e.g., `libudf`), and name it `Makefile`.

   Note that *path* is the directory in which you have installed ANSYS FLUENT, and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

3. In the library directory you just created in Step 1, make a directory that will store your source file and name it `src`.

4. Copy your source file (e.g., `myudf.c`) to the source directory (`/src`).

5. Copy `makefile.udf` from

$$path/\texttt{ansys\_inc/v120/fluent/fluent12.0.}\overset{\Downarrow}{x}\texttt{/src/makefile.udf}$$

   to the `/src` directory, and name it `makefile`.

   Note that *path* is the directory in which you have installed ANSYS FLUENT, and x is replaced by the appropriate number for the release (e.g., `9` for `fluent12.0.9`).

6. Identify the architecture name of the machine on which you are running (e.g., `ultra`). This can be done by either typing the command (`fluent-arch`) in the ANSYS FLUENT TUI window, or running the ANSYS FLUENT utility program `fluent_arch` at the command line of a UNIX shell.

   $\boxed{i}$ Note that if you are running a 64-bit version of ANSYS FLUENT the architecture name will have a `_64` appended to it (e.g., `ultra_64`).

7. In the library directory (e.g., `libudf`), create an architecture directory that is named after the architecture identifier determined in the previous step (e.g., `ultra`).

8. In the architecture directory, create directories named after the ANSYS FLUENT versions for which you want to build shared libraries (e.g., `ultra/2d` and `ultra/3d`). Possible versions are:

`2d` or `3d`	single-precision serial 2D or 3D
`2ddp` or `3ddp`	double-precision serial 2D or 3D
`2d_node` and `2d_host`	single-precision parallel 2D
`3d_node` and `3d_host`	single-precision parallel 3D
`2ddp_node` and `2ddp_host`	double-precision parallel 2D
`3ddp_node` and `3ddp_host`	double-precision parallel 3D

> *i* Note that you must create *two* build directories for each parallel version of the solver (two for the 3D version, two for the 2D double-precision version, etc.), regardless of the number of compute nodes.

5.3.2 Build the UDF Library

After you have set up the folder structure and put the files in the proper places, you can compile and build the shared library using the TUI.

Windows Systems

1. Using a text editor, edit every `user_nt.udf` file in each version folder to set the following parameters: `SOURCES`, `VERSION`, and `PARALLEL_NODE`.

`SOURCES =`	the user-defined source file(s) to be compiled. Use the prefix `$(SRC)` before each filename. For example, `$(SRC)udfexample.c` for one file, and `$(SRC)udfexample1.c $(SRC)udfexample2.c` for two files.
`VERSION =`	the version of the solver you are running which will be the name of the build folder where `user_nt.udf` is located. (`2d`, `3d`, `2ddp`, `3ddp`, `2d_host`, `2d_node`, `3d_host`, `3d_node`, `2ddp_host`, `2ddp_node`, `3ddp_host`, or `3ddp_node`).

PARALLEL_NODE =   the parallel communications library.
Specify `none` for a serial version of the solver or one of the following:
`smpi`: parallel using shared memory (for multiprocessor machines)
`vmpi`: parallel using shared memory or network with vendor MPI software
`net`: parallel using network communicator with RSHD software

> *i* If you are using a parallel version of the solver, be sure to edit *both* copies of `user_nt.udf` (the one in the host folder and the one in the node folder), and specify the appropriate `SOURCE`, `VERSION`, and `PARALLEL_NODE` in each file. Set `PARALLEL_NODE = none` for the host version and one of the other options `smpi, vmpi, net, nmpi` for the node version depending on which message passing method you are going to use.

An excerpt from a sample `user_nt.udf` file is shown below:

```
# Replace text in " " (and remove quotes)
#  | indicates a choice
#  note: $(SRC) is defined in the makefile

SOURCES = $(SRC)udfexample.c
VERSION = 2d
PARALLEL_NODE = none
```

2. In the Visual Studio command prompt window, go to each version folder (e.g., `\libudf\ntx86\2d\`), and type `nmake` as shown in the following example.

```
C:\users\user_name\work_dir\libudf\ntx86\2d>nmake
```

The following messages will be displayed:

```
Microsoft (R) Program Maintenance Utility Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.
cl /c /Za /DUDF_EXPORTING
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\ntx86\2d
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\cortex\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\client\src
```

```
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\tgrid\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\multiport\src
      ..\..\src\udfexample.c
Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

udfexample.c
# Generating udf_names.c because of makefile udfexample.obj
        cl /c /Za /DUDF_EXPORTING
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\ntx86\2d
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\cortex\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\client\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\tgrid\src
 -Ic:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\multiport\src
      udf_names.c
Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

udf_names.c
# Linking libudf.dll because of makefile user_nt.udf
udf_names.obj udfexample.obj
link  /Libpath:c:\Program Files\ANSYS Inc\v120\fluent\fluent12.0.9\ntx86\2d
      /dll
 /out:libudf.dl
l  udf_names.obj udfexample.obj   fl1209s.lib
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

   Creating library libudf.lib and object libudf.exp

C:\Program Files\ANSYS Inc\v120\fluent\ntbin\ntx86\libudf\ntx86\2d>
```

i   Note that if there are problems with the build, you can do a complete
rebuild by typing `nmake clean` and then `nmake` again.

UNIX and Linux Systems

1. Using a text editor, edit the file `makefile` in your `src` directory to set the following two parameters: `SOURCES` and `FLUENT_INC`.

`SOURCES =`	the name of your source file(s) (e.g., `udfexample.c`) Multiple sources can be specified by using a space delimiter (e.g., `udfexample1.c udfexample2.c`)
`FLUENT_INC =`	the path to your release directory

2. If your architecture is `irix6.5`, make the following additional change to the `makefile`.

    (a) Find the following line in the `makefile`:

        CFLAGS_IRIX6R10=   -KPIC -ansi -fullwarn -O -n32

    (b) Change `-ansi` to `-xansi`:

        CFLAGS_IRIX6R10=   -KPIC -xansi -fullwarn -O -n32

    For all other architectures, do not make any further changes to the `makefile`.

    An excerpt from a sample `makefile` is shown below:

```
#----------------------------------------------------------#
# makefile for user defined functions.
#
#----------------------------------------------------------#
#----------------------------------------------------------#
# User modifiable section.
#----------------------------------------------------------#
SOURCES= udfexample1.c
FLUENT_INC= /path/ansys_inc/v120/fluent
# Precompiled User Object files (for example .o files from .f
sources)
USER_OBJECTS=
```

```
#-------------------------------------------------------------#
# Build targets (do not modify below this line).
#-------------------------------------------------------------#
.
.
.
```

Note that in the previous example, *path* represents the directory where you installed ANSYS FLUENT.

3. In your library directory (e.g., `libudf`), execute the `Makefile` by typing a command that begins with `make` and includes the architecture of the machine you will run ANSYS FLUENT on, which you identified in a previous step. For example, for the Linux (`lnx86`) architecture type:

```
make "FLUENT_ARCH=lnx86"
```

ANSYS FLUENT will build a shared library for each version you created a directory for (Section 5.3.1: Set Up the Directory Structure) and will display messages about the compile/build process in the console. You can view the compilation history in the `log` file that is saved in your working directory.

For example, when compiling/building a shared library for a source file named `profile.c` and a UDF library named `libudf` on a Linux architecture, the console messages may include the following:

```
Working...
for d in lnx86[23]*; do \
   ( \
     cd $d; \
     for f in ../../src*.[ch] ../../src/makefile; do \
       if [ ! -f 'basename $f' ]; then \
         echo "# linking to " $f "in" $d; \
         ln -s $f .; \
       fi; \
     done; \
     echo ""; \
     echo "# building library in" $d; \
     make -k>makelog 2>&1; \
     cat makelog; \
   ) \
done
# linking to ...   myudf.c in lnx86/2d
```

```
# building library in lnx86/2d
make[1]: Entering directory ..../udf_names.c
# Generating udf_names
make[2]: Entering directory ..../profile.c
make libudf.so ...
# Compiling udf_names.o ...
# Compiling profile.o ...
# Linking libudf.so ...
make[2]: Leaving directory ..../udf_names.c
make[1]: Leaving directory ..../profile.c

You can also see the 'log'-file in
the working directory for compilation history
Done.
```

5.3.3 Load the UDF Library

You can load the shared library you compiled and built using the GUI from the Compiled UDFs dialog box or the UDF Library Manager dialog box. Follow the procedure outlined in Step 9 of Section 5.2: Compiling a UDF Using the GUI or in Section 5.5: Load and Unload Libraries Using the UDF Library Manager Dialog Box, respectively.

5.4 Link Precompiled Object Files From Non-ANSYS FLUENT Sources

ANSYS FLUENT allows you to build a shared library for precompiled object files that are derived from external sources using the text user interface (TUI) option. For example, you can link precompiled objects derived from FORTRAN sources (`.o` objects from `.f` sources) to ANSYS FLUENT for use by a UDF. The following sections describe the procedures for doing this on a Windows system and a UNIX/Linux system.

Windows Systems

1. Follow the procedure for setting up the folder structure described in Section Section 5.3.1: Set Up the Directory Structure.

2. Copy your precompiled object files (e.g., `myobject1.obj myobject2.obj`) to all of the architecture/version folders you created in Step 1 (e.g., `ntx86/2d`, `ntx86/3d`).

   i The object files should be compiled using similar flags to those used by ANSYS FLUENT (e.g., `/c /Za`).

3. Using a text editor, edit the `user_nt.udf` files in each architecture/version folder.

UNIX and Linux Systems

1. Follow the procedure for setting up the directory structure described in Section 5.3.1: Set Up the Directory Structure.

2. Copy your precompiled object files (e.g., `myobject1.o myobject2.o`) to all of the architecture/version directories you created in Step 1 (e.g., `ultra/2d` and `ultra/3d`).

   i   The object files should be compiled using similar flags to those used for ANSYS FLUENT. Common flags used by ANSYS FLUENT are: `-KPIC`, `-O`, and `-ansi` which often have equivalents such as `-fpic`, `-O3`, and `-xansi`.

3. Using a text editor, edit the file `makefile` in your `src` directory to set the following three parameters: `SOURCES`, `FLUENT_INC`, and `USER_OBJECTS`.

`SOURCES =`	Put the names of your UDF C files here. They will be calling the functions in the User Objects.
`FLUENT_INC =`	the path to your release directory.
`USER_OBJECTS =`	the precompiled object file(s) that you want to build a shared library for (e.g., `myobject1.o`). Use a space delimiter to specify multiple object files (e.g., `myobject1.o myobject2.o`).

   An excerpt from a sample `makefile` is shown below:

   ```
   #----------------------------------------------------------#
   # makefile for user defined functions
   #
   #----------------------------------------------------------#
   #----------------------------------------------------------#
   # User modifiable section.
   #----------------------------------------------------------#
   SOURCES=udf_source1.c
   FLUENT_INC= /path/ansys_inc/v120/fluent
   # Precompiled User Object files (for example .o files from .f
   sources)
   USER_OBJECTS= myobject1.o myobject2.o
   ```

```
#------------------------------------------------------------#
# Build targets (do not modify below this line).
#------------------------------------------------------------#
.
.
.
```

Note that in the previous example, *path* represents the directory where you installed ANSYS FLUENT.

4. In your library directory (e.g., `libudf`), execute the `Makefile` by typing a command that begins with `make` and includes the architecture of the machine on which you will run ANSYS FLUENT, which you identified in a previous step (e.g., `ultra`).

```
make  "FLUENT_ARCH=ultra"
```

The following messages will be displayed:

```
# linking to ../../src/makefile in ultra/2d
# building library in ultra/2d
# linking to ../../src/makefile in ultra/3d
# building library in ultra/3d
```

5.4.1 Example - Link Precompiled Objects to ANSYS FLUENT

The following example demonstrates the linking of a FORTRAN object file `test.o` to ANSYS FLUENT, for use in a UDF named `test_use.c`. This particular UDF is not a practical application but has rather been designed to demonstrate the functionality. It uses data from a FORTRAN-derived object file to display parameters that are passed to the C function named `fort_test`. This on-demand UDF, when executed from the User-Defined Function Hooks dialog box, displays the values of the FORTRAN parameters and the common block and common complex numbers that are computed by the UDF, using the FORTRAN parameters.

> **i** Note that the names of the functions and data structures have been changed from the capital form in FORTRAN (e.g., `ADDAB` is changed to `addab_`). This name "mangling" is done by the compiler and is strongly system-dependent. Note also that functions returning complex numbers have different forms on different machine types, since C can return only single values and not structures. Consult your system and compiler manuals for details.

1. In the first step of this example, a FORTRAN source file named `test.f` is compiled and the resulting object file (`test.o`) is placed in the shared library folder for the `ultra/2d` version.

   ```
   libudf/ultra/2d
   ```

   The source listing for `test.f` is shown below.

   ```
   C  FORTRAN function
   C  test.f
   C
   C  compile to .o file using:
   C  f77 -KPIC -n32 -O -c test.f (irix6 & suns)

         REAL*8 FUNCTION ADDAB(A,B,C)

         REAL A
         REAL*8 B
         REAL*8 YCOM
         COMPLEX ZCOM
         INTEGER C
         INTEGER SIZE

         COMMON //SIZE,ARRAY(10)
         COMMON /TSTCOM/ICOM,XCOM,YCOM,ZCOM

         ICOM=C
         XCOM=A
         YCOM=B
         ZCOM=CMPLX(A,REAL(B))

         SIZE=10
         DO 100 I=1,SIZE
          ARRAY(I)=I*A
   100   CONTINUE

         ADDAB=(A*C)*B
         END

         COMPLEX FUNCTION CCMPLX(A,B)

         REAL A,B
   ```

```
         CCMPLX=CMPLX(A,B)
         END
```

2. The UDF C source file named `test_use.c` is placed in the source folder for the `ultra/2d` version:

   src/ultra/2d

   The source listing for `test_use.c` is as follows.

```
#include "udf.h"

#if defined(_WIN32)
  /*  Visual Fortran makes uppercase functions provide lowercase
      mapping to be compatible with UNIX code  */
#  define addab_ ADDAB
#endif

typedef struct {float r,i;} Complex;
typedef struct {double r,i;} DComplex;
typedef struct {long double r,i;} QComplex; /* FORTRAN QUAD
                                               PRECISION */

/* FORTRAN FUNCTION */
extern double addab_(float *a,double *b,int *c);

/* NOTE on SUN machines that FORTRAN functions returning a complex
   number are actually implemented as void but with an extra
   initial argument.*/

extern void ccmplx_(Complex *z,float *a,float *b);
extern void qcmplx_(QComplex *z,float *a,float *b);

/* BLANK COMMON BLOCK */
extern struct
{
 int size;
 float array[10];
} _BLNK__;
```

```
/* FORTRAN NAMED COMMON BLOCK */
extern struct
{
 int int_c;
 float float_a;
 double double_b;
 float cmplx_r;
 float cmplx_i;
} tstcom_;

DEFINE_ON_DEMAND(fort_test)
{
 float a=3.0,float_b;
 double d,b=1.5;
 int i,c=2;
 Complex z;
 QComplex qz;

 d = addab_(&a,&b,&c);
 Message("\n\nFortran code gives (%f * %d) * %f = %f\n",a,c,b,d);
 Message("Common Block TSTCOM set to: %g %g %d\n",
         tstcom_.float_a,tstcom_.double_b,tstcom_.int_c);
 Message("Common Complex Number is (%f + %fj)\n",
         tstcom_.cmplx_r,tstcom_.cmplx_i);
 Message("BLANK Common Block has an array of size %d:
                                      \n",_BLNK__.size);
 for (i=0; i <_BLNK__.size ; i++)
   {
     Message("array[%d] = %g\n",i,_BLNK__.array[i]);
   }

 float_b=(float)b;
 ccmplx_(&z,&a,&float_b);
 Message("Function CCMPLX returns Complex Number:
                                      (%g + %gj)\n",z.r,z.i);
 qcmplx_(&qz,&a,&float_b);
 Message("Function QCMPLX returns Complex Number:
                                      (%g + %gj)\n",qz.r,qz.i);
}
```

3. The `makefile` is then modified to specify the UDF C source file (`test_use.c`) and the external object file (`test.o`) as shown below.

```
#---------------------------------------------------------------#
# User modifiable section.
#---------------------------------------------------------------#
SOURCES= test_use.c
FLUENT_INC= /path/ansys_inc/v120/fluent
# Precompiled User Object files (for example .o files from .f
sources)
USER_OBJECTS= test.o
```

Note that in the previous example, *path* represents the directory where you installed ANSYS FLUENT.

4. Finally, the `Makefile` is executed by issuing the following command in the `libudf` folder:

```
make "FLUENT_ARCH=ultra"
```

5.5   Load and Unload Libraries Using the UDF Library Manager **Dialog Box**

You can use the UDF Library Manager dialog box to load and unload multiple shared libraries in ANSYS FLUENT.

Load the UDF Library

To load a UDF library in ANSYS FLUENT, open the UDF Library Manager dialog box (Figure 5.5.1).

Define $\longrightarrow$ User-Defined $\longrightarrow$ Functions $\longrightarrow$ Manage...
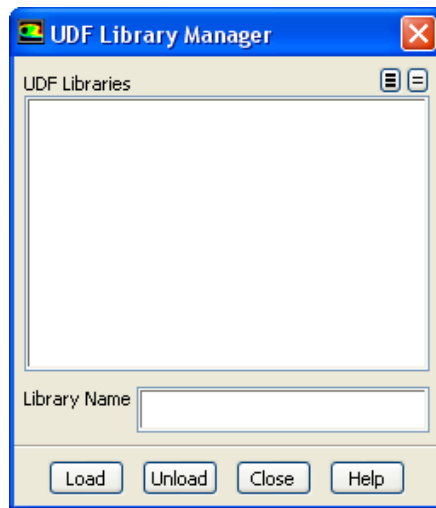
Figure 5.5.1: The UDF Library Manager Dialog Box

In the UDF Library Manager dialog box, type the name of the shared library in the Library Name field and click Load (Figure 5.5.1).

A message will be displayed in the console providing a status of the load process. For example:

```
Opening library "libudf"...
Library "libudf\ntx86\3d\libudf.dll" opened
        inlet_x_velocity
Done.
```

indicates that the shared library named libudf was successfully loaded and contains one UDF named inlet_x_velocity. In the UDF Library Manager dialog box, the library name (e.g., libudf) will be added under UDF Libraries. Repeat this step to load additional libraries.

Unload the UDF Library

To unload a UDF library in ANSYS FLUENT, open the UDF Library Manager dialog box (Figure 5.5.2).

Define ⟶ User-Defined ⟶ Functions ⟶Manage...
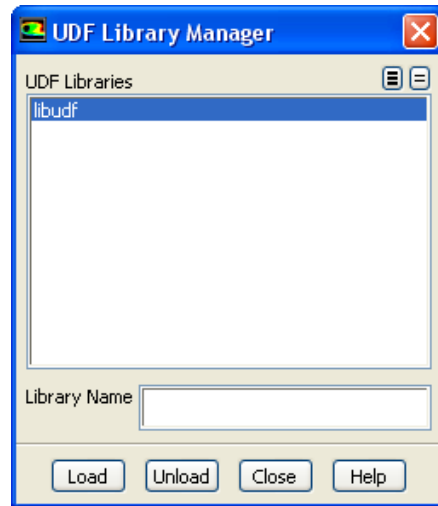
Figure 5.5.2: The UDF Library Manager Dialog Box

In the UDF Library Manager dialog box, highlight the shared library name (e.g., `libudf`) that is listed under UDF Libraries (or type the Library Name) and click Unload (Figure 5.5.2).

After it is unloaded, the library (e.g., `libudf`) will be removed from the UDF Libraries list in the dialog box. Repeat this step to unload additional libraries.

5.6 Common Errors When Building and Loading a UDF Library

A common compiler error occurs when you forget to put an `#include "udf.h"` statement at the beginning of your source file. You'll get a long list of compiler error messages that include illegal declarations of variables. Similarly, if your function requires an auxiliary header file (e.g., `sg_pdf.h`) and you forgot to include it, you'll get a similar compiler error message.

Another common error occurs when the argument list for a `DEFINE` statement is placed on multiple lines. (All `DEFINE` macro arguments must be listed on the same line in a C file.) The compiler will typically not report any error message but it will report a single warning message in the log file to indicate that this occurred:

```
warning:  no newline at end of file
```

If your compiled UDF library loads successfully then each function contained within the library will be reported to the console (and log file). For example, if you built a shared library named `libudf` containing two user-defined functions `superfluid_density` and `speed_sound`, a successful library load (on a Linux machine) will result in the following message reported to the console (and log file) for a Linux machine:

```
Opening library "libudf"...
Library "libudf/lnx86/3d/libudf.so" opened
        superfluid_density
        speed_sound
Done.
```

If, instead, no function names are listed, then it is likely that your source file did not successfully compile. In this case, you'll need to consult the log to view the compilation history, and debug your function(s). Note that you'll need to unload the UDF library using the UDF Library Manager dialog box before you reload the debugged version of your library.

Another common error occurs when you try to read a case file that was saved with a shared library, and that shared library has subsequently been moved to another location. In this case, the following error will be reported to the console (and log file) on a Linux machine:

```
Opening library "libudf"...
Error: open_udf_library: couldn't open library: libudf/ln86/2d/libudf.so
```

Similarly, you will get an error message when you try to load a shared library before it has been built.

```
Opening library "libudf"...
Error: open_udf_library: No such file or directory
```

Windows Parallel

If you are trying to load a compiled UDF while running ANSYS FLUENT in network parallel, you may receive this error:

```
Error: open_udf_library: The system cannot find the path specified
```

This error occurs because the other computer(s) on the cluster cannot "see" the UDF through the network. To remedy this, you will need to 1) modify the environment variables on the computer where the compiled UDF, case, and data files reside; and 2) share the folder where the files reside. See Section 5.2: Compiling a UDF Using the GUI for details on file sharing or contact ANSYS FLUENT installation support for additional assistance.

There are instances when ANSYS FLUENT can hang when trying to read a compiled UDF using network parallel as a result of a network communicator problem. Contact ANSYS FLUENT installation support for details.

You may receive an error message when you invoke the command `nmake` if you have the wrong compiler installed or if you have not launched the Visual Studio Command Prompt prior to building the UDF. See Section 5.1.2: Compilers and Section 5.2: Compiling a UDF Using the GUI for details or contact ANSYS FLUENT installation support for further assistance.

5.7   Special Considerations for Parallel ANSYS FLUENT

If you are running serial or parallel ANSYS FLUENT on a Windows system and intend to compile a UDF, then you must have Microsoft Visual Studio installed on your machine, preferably on the `C:` drive.

Also note that if you have compiled a UDF while running ANSYS FLUENT on a Windows parallel network, you *must* 'share' the folder where the UDF is located so that all computers on the cluster can see this folder. To share the folder in which the case, data, and compiled UDF reside, using the Windows Explorer right-click on the folder, choose Sharing... from the menu, click Share this folder, and then click OK.

| *i* | If you forget to enable the sharing option for the folder using the Windows Explorer, then ANSYS FLUENT will hang when you try to load the library in the Compiled UDFs dialog box. |

See Section 5.6: Common Errors When Building and Loading a UDF Library for a list of errors you can encounter that are specific to Windows parallel.

Chapter 6. Hooking UDFs to ANSYS FLUENT

After you have interpreted or compiled your UDF using the methods described in Chapters 4 and 5, respectively, you are ready to hook the function to ANSYS FLUENT using a graphic interface dialog box. After it is hooked, the function will be utilized in your ANSYS FLUENT model. Details about hooking a UDF to ANSYS FLUENT can be found in the following sections. Note that these sections relate to corresponding sections in Chapter 2: DEFINE Macros.

- Section 6.1: Hooking General Purpose UDFs

- Section 6.2: Hooking Model-Specific UDFs

- Section 6.3: Hooking Multiphase UDFs

- Section 6.4: Hooking Discrete Phase Model (DPM) UDFs

- Section 6.5: Hooking Dynamic Mesh UDFs

- Section 6.6: Hooking User-Defined Scalar (UDS) Transport Equation UDFs

- Section 6.7: Common Errors While Hooking a UDF to ANSYS FLUENT

6.1 Hooking General Purpose UDFs

This section contains methods for hooking general purpose UDFs to ANSYS FLUENT. General purpose UDFs are those that have been defined using macros described in Section 2.2: General Purpose DEFINE Macros and then interpreted or compiled and loaded using methods described in Chapters 4 or 5, respectively.

6.1.1 Hooking `DEFINE_ADJUST` **UDFs**

After you interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_ADJUST` UDF, the name of the function you supplied as a `DEFINE` macro argument can be hooked using the User-Defined Function Hooks dialog box (Figure 6.1.1). Note that you can hook multiple adjust UDFs to your model, if desired.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.1.1)

Define ⟶ User-Defined ⟶ Function Hooks...
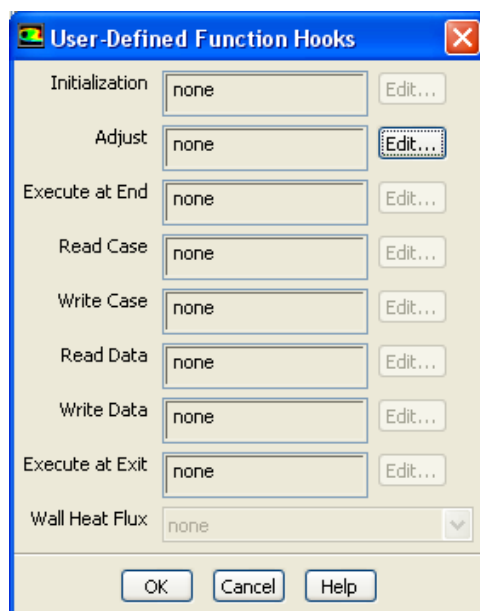
Figure 6.1.1: The User-Defined Function Hooks Dialog Box

Click the Edit... button next to Adjust to open the Adjust Functions dialog box (Figure 6.1.2).
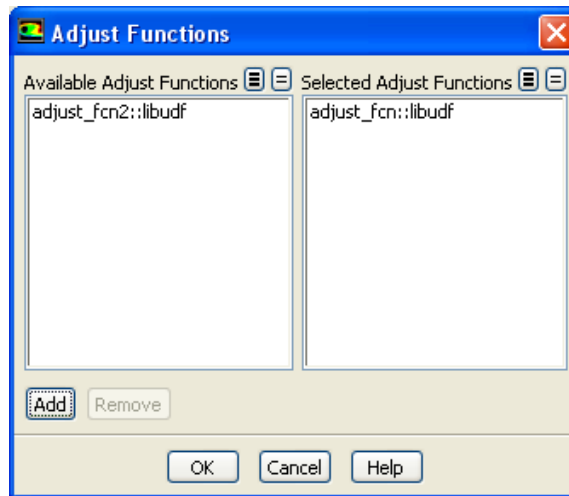
Figure 6.1.2: The Adjust Functions Dialog Box

Select the function(s) you wish to hook to your model from the Available Adjust Functions list. Click Add and then OK to close the dialog box. The name of the function you selected will be displayed in the Adjust field of the User-Defined Function Hooks dialog box. If you select more than one function, the number will be displayed (e.g., 2 selected). Click OK in the User-Defined Function Hooks dialog box to apply the settings.

See Section 2.2.1: DEFINE_ADJUST for details about defining adjust functions using the DEFINE_ADJUST macro.

6.1.2 Hooking DEFINE_DELTAT UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DELTAT UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Adaptive Time Step Settings dialog box (Figure 6.1.3) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, select Transient from the Time list in the Solver group box of the General task page.

⇨ General ⟶ ⊙ Transient

Then open the Run Calculation task page.

⇨ Run Calculation

Select Adaptive in the Time Stepping Method drop-down list, and click the Settings... button to open the Adaptive Time Step Settings dialog box (Figure 6.1.3).
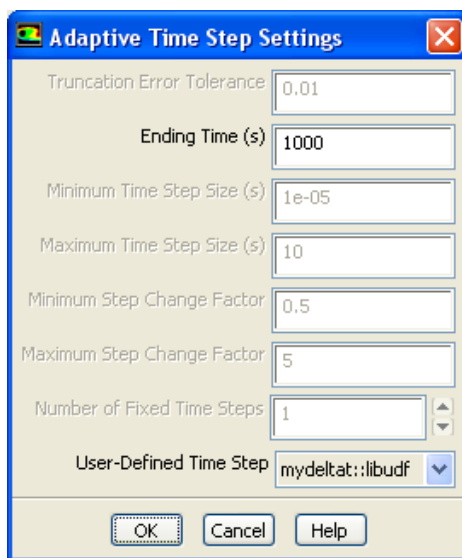
Figure 6.1.3: The Adaptive Time Step Settings Dialog Box

Select the function name (e.g., mydeltat::libudf) in the User-Defined Time Step drop-down list and click OK.

> **i** Note that when you are using the VOF Multiphase Model, you will need to select Variable as the Time Stepping Method in the Run Calculation task page, and then hook the time step UDF in the Variable Time Step Setting dialog box in a similar manner to the Adaptive Time Step Settings dialog box.

See Section 2.2.2: DEFINE_DELTAT for details about defining DEFINE_DELTAT functions.

6.1.3 Hooking DEFINE_EXECUTE_AT_END **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_EXECUTE_AT_END UDF, it is ready to be hooked to ANSYS FLUENT. Note that you can hook multiple at-end UDFs to your model, if desired.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.1.4)

 Define ⟶ User-Defined ⟶Function Hooks...
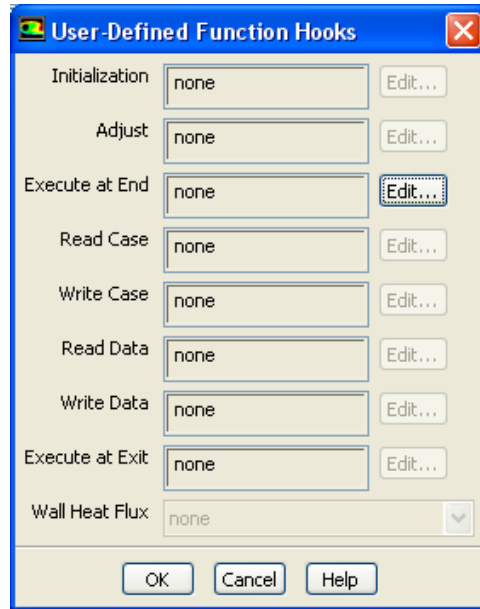
Figure 6.1.4: The User-Defined Function Hooks Dialog Box

Click the Edit... button next to Execute At End to open the Execute At End Functions dialog box (Figure 6.1.5).
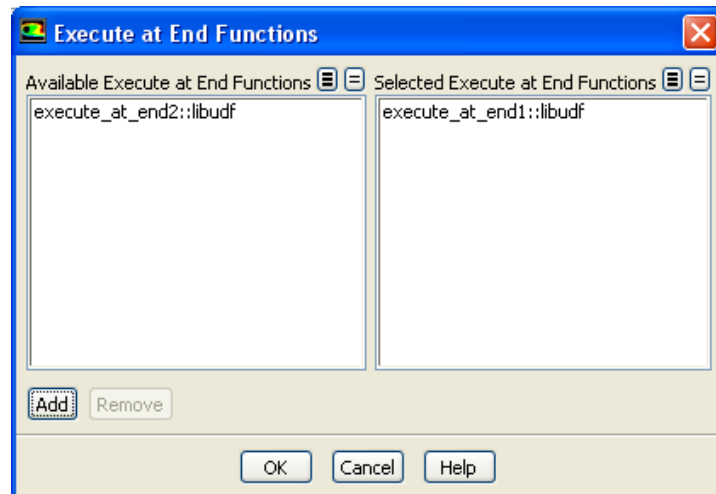
Figure 6.1.5: The Execute At End Functions Dialog Box

Select the function(s) you wish to hook to your model from the Available Execute at End Functions list. Click Add and then OK to close the dialog box. The name of the function you selected will be displayed in the Execute at End field of the User-Defined Function Hooks dialog box. If you select more than one function, the number will be displayed (e.g., 2 selected). Click OK in the User-Defined Function Hooks dialog box to apply the settings.

See Section 2.2.3: DEFINE_EXECUTE_AT_END for details about defining DEFINE_EXECUTE_AT_END functions.

6.1.4 Hooking DEFINE_EXECUTE_AT_EXIT **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_EXECUTE_AT_EXIT UDF, it is ready to be hooked to ANSYS FLUENT. Note that you can hook multiple at-exit UDFs to your model, if desired.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.1.6)

Define ⟶ User-Defined ⟶Function Hooks...
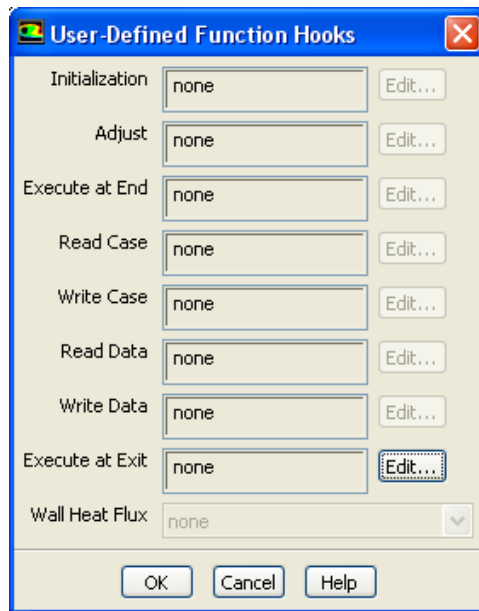
Figure 6.1.6: The User-Defined Function Hooks Dialog Box

Click the Edit... button next to Execute At Exit to open the Execute At Exit Functions dialog box (Figure 6.1.7).

Select the function(s) you wish to hook to your model from the Available Execute At End Functions list. Click Add and then OK to close the dialog box. The name of the function you selected will be displayed in the Execute At Exit field of the User-Defined Function Hooks dialog box. If you select more than one function, the number will be displayed (e.g., 2 selected). Click OK in the User-Defined Function Hooks dialog box to apply the settings.

See Section 2.2.4: DEFINE_EXECUTE_AT_EXIT for details about defining DEFINE_EXECUTE_AT_EXIT functions.
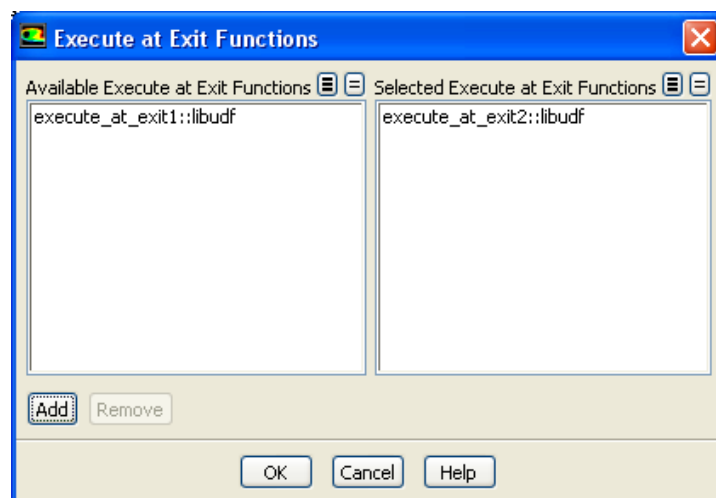
Figure 6.1.7: The **Execute At Exit Functions** Dialog Box

6.1.5   Hooking DEFINE_INIT **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_INIT UDF, it is ready to be hooked to ANSYS FLUENT. Note that you can hook multiple initialization UDFs to your model, if desired.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.1.8)

Define ⟶ User-Defined ⟶Function Hooks...
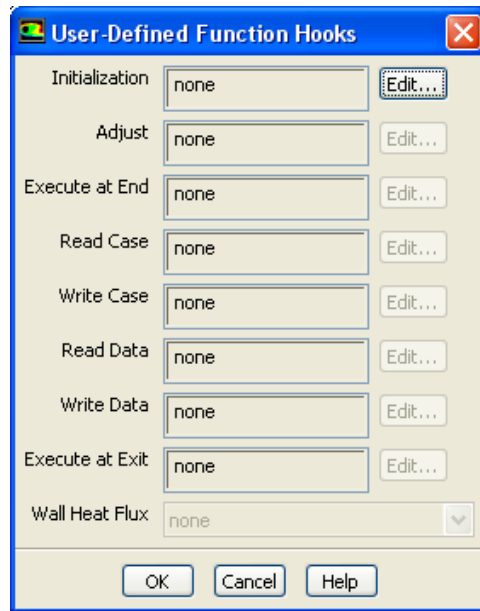
Figure 6.1.8: The User-Defined Function Hooks Dialog Box

Click the Edit... button next to Initialization to open the Initialization Functions dialog box (Figure 6.1.9).
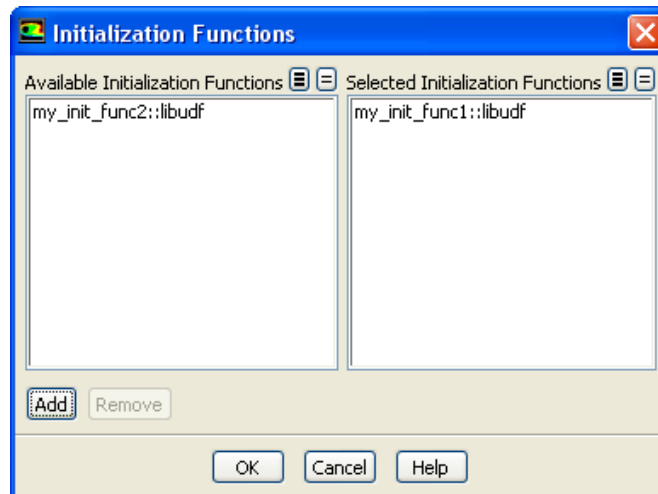
Figure 6.1.9: The Initialization Functions Dialog Box

Select the function(s) you wish to hook to your model from the Available Initialization Functions list. Click Add and then OK to close the dialog box. The name of the function you selected will be displayed in the Initialization field of the User-Defined Function Hooks dialog box. If you select more than one function, the number will be displayed (e.g., 2 selected). Click OK in the User-Defined Function Hooks dialog box to apply the settings.

See Section 2.2.8: DEFINE_INIT for details about defining DEFINE_INIT functions.

6.1.6 Hooking DEFINE_ON_DEMAND **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_ON_DEMAND UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Execute On Demand dialog box (Figure 6.1.10) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, open the Execute On Demand dialog box.

Define ⟶ User-Defined ⟶Execute On Demand...
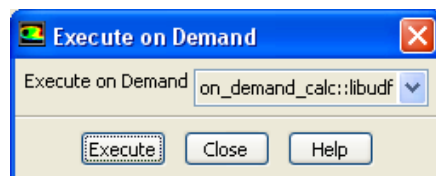
Figure 6.1.10: The Execute On Demand Dialog Box

Select the function name (e.g., on_demand_calc::libudf) in the Execute On Demand drop-down list and click Execute. ANSYS FLUENT will execute the UDF immediately. Click Close to close the dialog box.

See Section 2.2.9: DEFINE_ON_DEMAND for details about defining DEFINE_ON_DEMAND functions.

6.1.7 Hooking DEFINE_RW_FILE UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_RW_FILE UDF, it is ready to be hooked to ANSYS FLUENT. Note that you can hook multiple read/write file UDFs to your model, if desired.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.1.11)

Define ⟶ User-Defined ⟶Function Hooks...
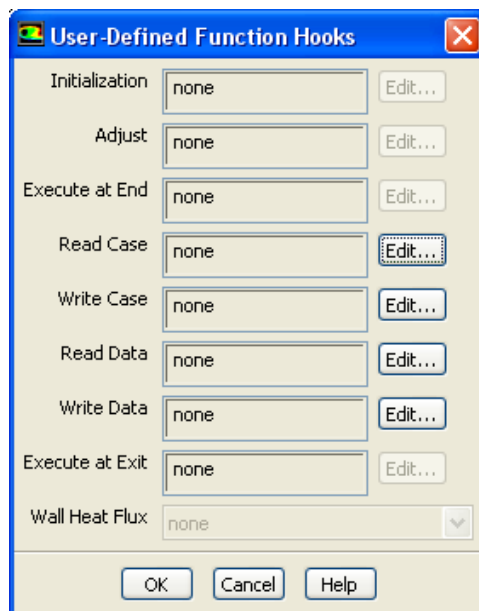
Figure 6.1.11: The User-Defined Function Hooks Dialog Box

You have the choice of hooking a UDF to read and write a case and data file. Below is a description of what each function does.

- Read Case is called when you read a case file into ANSYS FLUENT. It will specify the customized section that is to be read from the case file.

- Write Case is called when you write a case file from ANSYS FLUENT. It will specify the customized section that is to be written to the case file.

- Read Data is called when you read a data file into ANSYS FLUENT. It will specify the customized section that is to be read from the data file.

- Write Data is called when you write a data file from ANSYS FLUENT. It will specify the customized section that is to be written to the data file.

To hook a read case file UDF, for example, click the Edit... button next to Read Case to open the Read Case Functions dialog box (Figure 6.1.12).
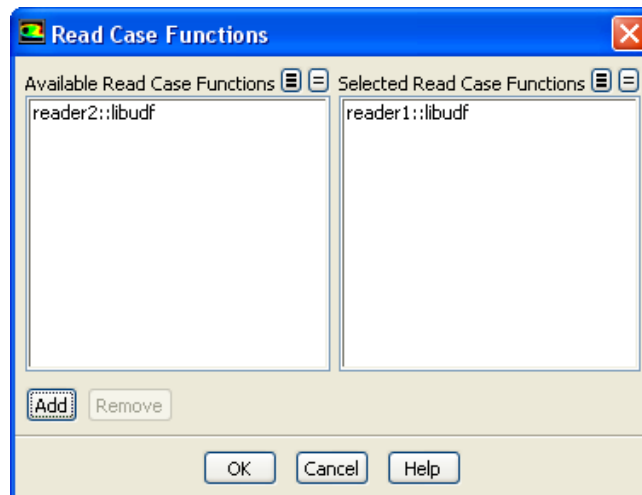
Figure 6.1.12: The Read Case Functions Dialog Box

Select the function(s) you wish to hook to your model from the Available Read Case Functions list. Click Add and then OK to close the dialog box. The name of the function you selected will be displayed in the Read Case field of the User-Defined Function Hooks dialog box. If you select more than one function, the number will be displayed (e.g., 2 selected). Click OK in the User-Defined Function Hooks dialog box to apply the settings.

See Section 2.2.10: DEFINE_RW_FILE for details about defining DEFINE_RW_FILE functions.

6.1.8   User-Defined Memory Storage

You can store values computed by your UDF in memory so that they can be retrieved later, either by a UDF or for postprocessing within ANSYS FLUENT. In order to have access to this memory, you will need to allocate memory by specifying the **Number of User-Defined Memory Locations** in the **User-Defined Memory** dialog box (Figure 6.1.13).

Define ⟶ User-Defined ⟶ Memory...
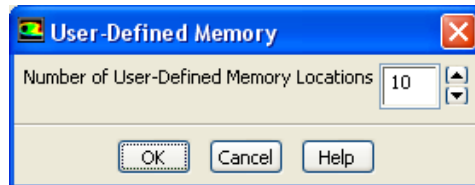
Figure 6.1.13: The User-Defined Memory Dialog Box

The macros `C_UDMI` or `F_UDMI` can be used in your UDF to access a particular user-defined memory location in a cell or face, respectively. See Sections 3.2.3 and 3.2.4 for details.

Field values that have been stored in user-defined memory will be saved to the data file when you next write one. These fields will also appear in the User Defined Memory... category in the drop-down lists in ANSYS FLUENT's postprocessing dialog boxes. They will be named User Memory 0, User Memory 1, etc., based on the memory location index. The total number of memory locations is limited to 500. For large numbers of user-defined memory locations, system memory requirements will increase.

6.2   Hooking Model-Specific UDFs

This section contains methods for hooking model-specific UDFs to ANSYS FLUENT that have been defined using `DEFINE` macros found in Section 2.3: Model-Specific `DEFINE` Macros, and interpreted or compiled using methods described in Chapters 4 or 5, respectively.

6.2.1   Hooking `DEFINE_CHEM_STEP` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_CHEM_STEP` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.1) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first set up the species transport and combustion models in the Species Model dialog box.

⟳ Models ⟶ ☰ Species ⟶ Edit...

Note that chemistry step UDFs may only be used with the laminar finite-rate model (with stiff chemistry enabled), the EDC model, or the PDF Transport model. Therefore, you must use one of the following groups of settings in the Species Model dialog box:

- To enable the laminar finite-rate model, select Species Transport, enable Volumetric in the Reactions group box, select Laminar Finite-Rate in the Turbulence-Chemistry Interaction group box, and enable Stiff Chemistry Solver in the Options group box.

- To enable the EDC model, select Species Transport, enable Volumetric in the Reactions group box, and select Eddy-Dissipation Concept in the Turbulence-Chemistry Interaction group box.

- To enable the PDF Transport model, select Composition PDF Transport and enable Volumetric in the Reactions group box.

Next, open the User-Defined Function Hooks dialog box. (Figure 6.2.1)

Define $\longrightarrow$ User-Defined $\longrightarrow$ Function Hooks...
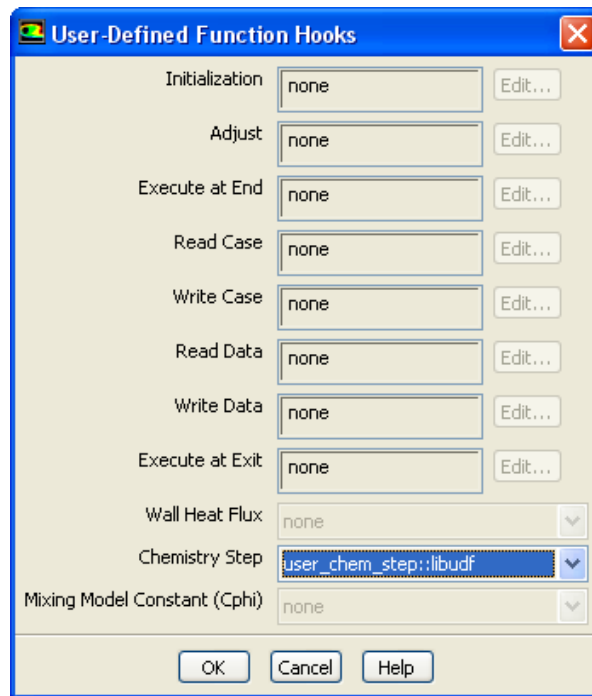
Figure 6.2.1: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., user_chem_step::libudf) in the Chemistry Step drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.1: DEFINE_CHEM_STEP for details about defining DEFINE_CHEM_STEP functions.

6.2.2 Hooking `DEFINE_CPHI` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_CPHI` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.2) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.2.2)

Define ⟶ User-Defined ⟶Function Hooks...
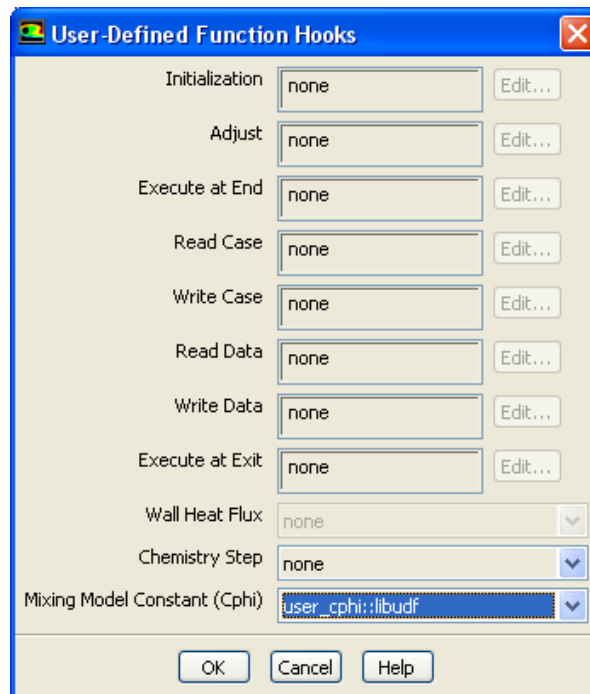
Figure 6.2.2: The User-Defined Function Hooks Dialog Box

i EDC or PDF Transport models must be enabled to hook the mixing model constant Cphi UDFs.

Select the function name (e.g., user_cphi::libudf) from the drop-down list for Mixing Model Constant (Cphi), and click OK.

See Section 2.3.2: `DEFINE_CPHI` for details about defining `DEFINE_CPHI` functions.

6.2.3 Hooking DEFINE_DIFFUSIVITY UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DIFFUSIVITY UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first open the Materials task page.

⇕ Materials

Make a selection in the Materials list and click the Create/Edit... button to open the appropriate Create/Edit Materials dialog box (Figure 6.2.3).
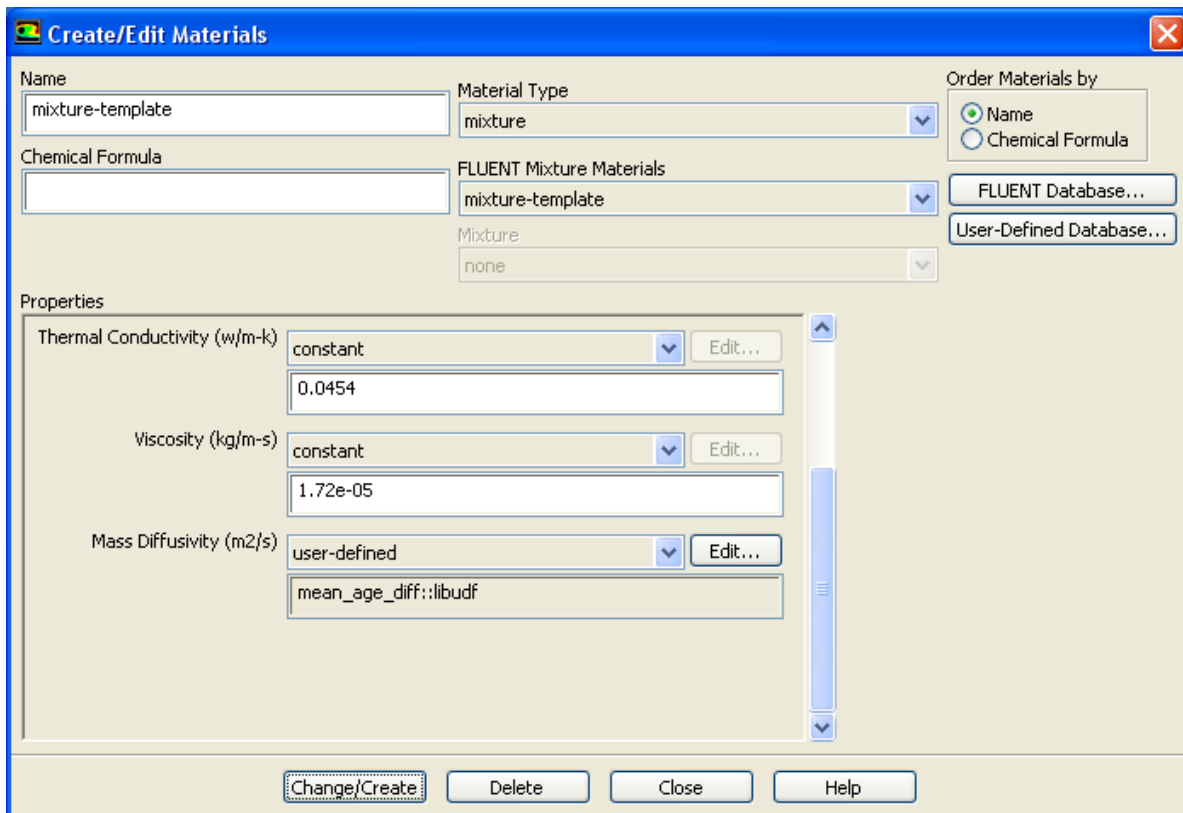
Figure 6.2.3: The Create/Edit Materials Dialog Box

You then have the following options:

- To hook a mass diffusivity UDF for the species transport equations, select user-defined from the Mass Diffusivity drop-down list of the Create/Edit Materials dialog box (Figure 6.2.3). The User-Defined Functions dialog box (Figure 6.2.4) will open.
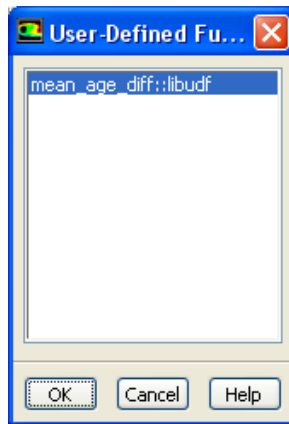
Figure 6.2.4: The User-Defined Functions Dialog Box

Select the name of your UDF (e.g., mean_age_diff::libudf) and click OK in the User-Defined Functions dialog box. The name will then be displayed in the field below the Mass Diffusivity drop-down list in the Create/Edit Materials dialog box. Click Change/Create to save your settings.

- To hook a single diffusion coefficient UDF so that it applies to all UDS equations, first set the number and options of the user-defined scalars using the User-Defined Scalars dialog box.

  Define $\longrightarrow$ User-Defined $\longrightarrow$ Scalars...

  Then, select user-defined from the UDS Diffusivity drop-down list in the Create/Edit Materials dialog box (Figure 6.2.5).

  Next, select the name of your UDF (e.g., mean_age_diff::libudf) in the User-Defined Functions dialog box that opens (Figure 6.2.4) and click OK. The name will then be displayed in the field below the UDS Diffusivity drop-down list in the Create/Edit Materials dialog box. Click Change/Create to save your settings.

See Section 2.3.3: DEFINE_DIFFUSIVITY for details about defining DEFINE_DIFFUSIVITY UDFs and the User's Guide for general information about UDS diffusivity.
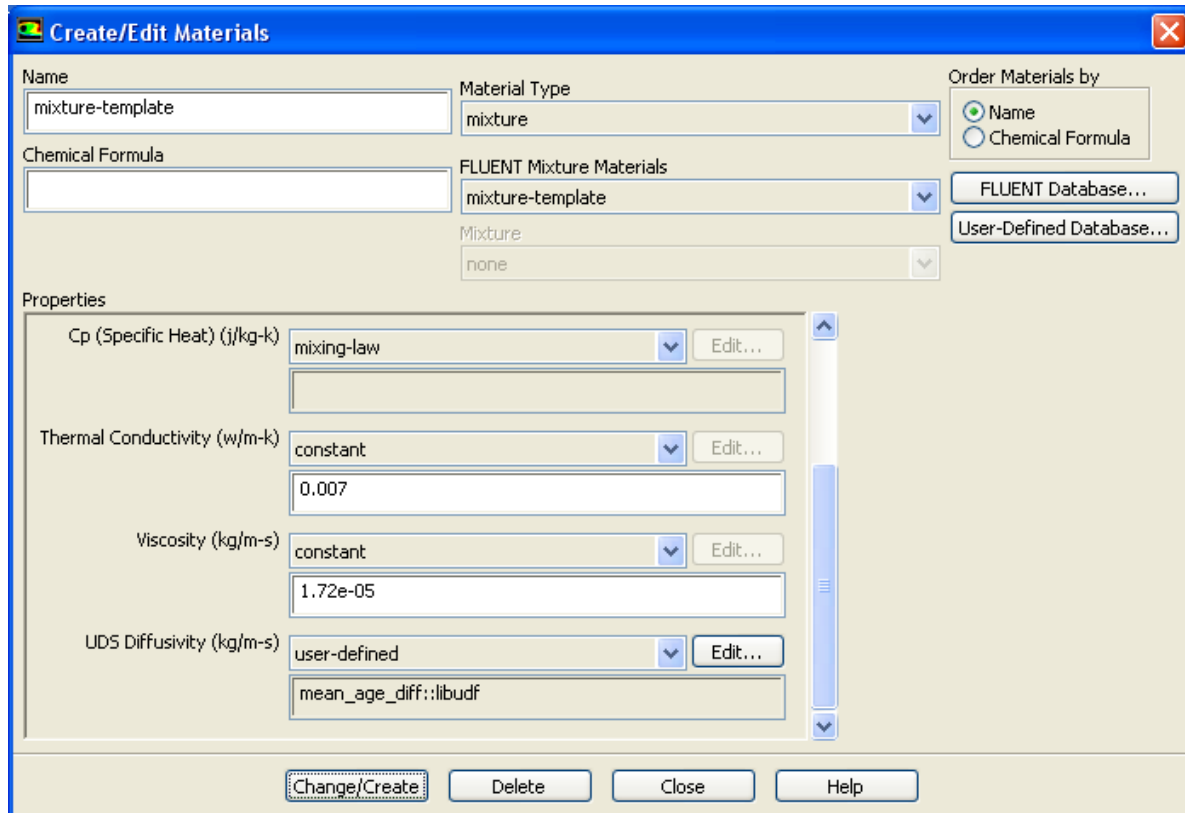
Figure 6.2.5: The **Create/Edit Materials** Dialog Box

6.2.4 Hooking DEFINE_DOM_DIFFUSE_REFLECTIVITY UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DOM_DIFFUSE_REFLECTIVITY UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.6) in ANSYS FLUENT.

$\boxed{i}$ The discrete ordinates (DO) radiation model must be enabled from the Radiation Model dialog box.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.2.6)

$\boxed{\text{Define}}\longrightarrow\boxed{\text{User-Defined}}\longrightarrow\text{Function Hooks...}$
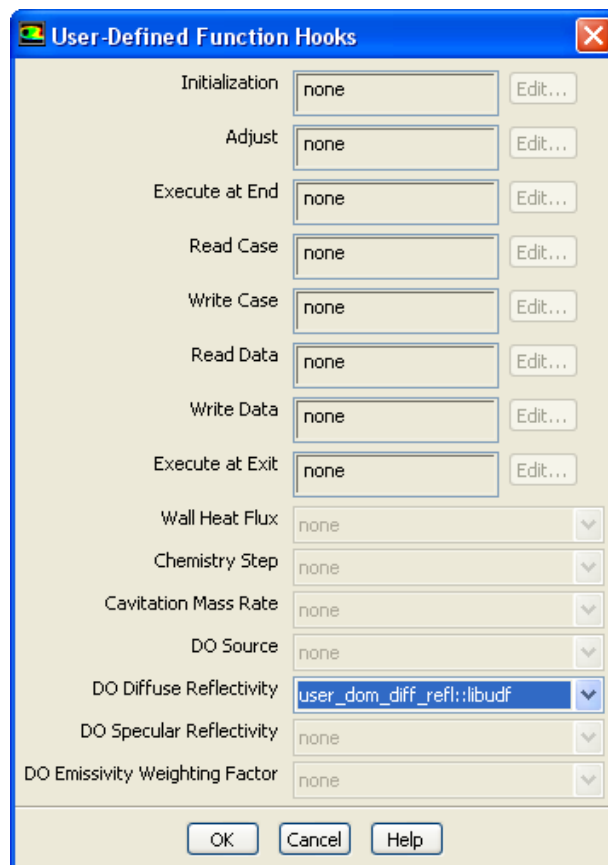
Figure 6.2.6: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., user_dom_diff_refl::libudf) in the DO Diffuse Reflectivity drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.4: `DEFINE_DOM_DIFFUSE_REFLECTIVITY` for details about `DEFINE_DOM_DIFFUSE_REFLECTIVITY` functions.

6.2.5 **Hooking** `DEFINE_DOM_SOURCE` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DOM_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.7) in ANSYS FLUENT.

i The discrete ordinates (DO) radiation model must be enabled from the Radiation Model dialog box.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.2.7)

Define ⟶ User-Defined ⟶ Function Hooks...
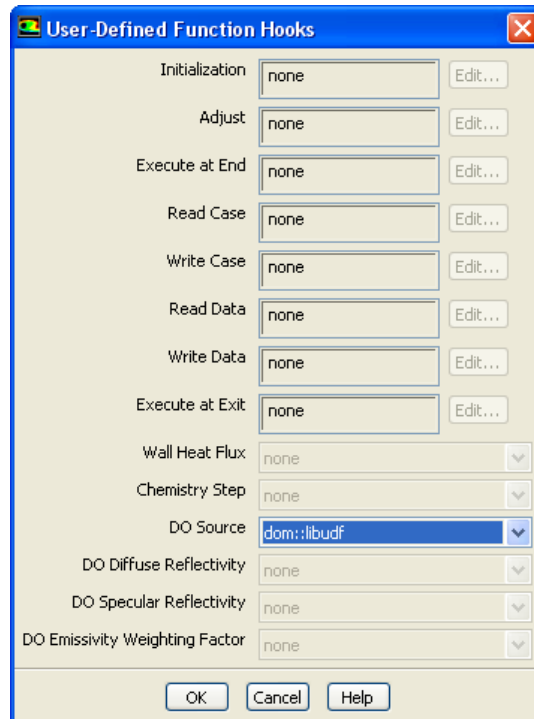
Figure 6.2.7: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., **dom::libudf**) in the **DO Source** drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.5: `DEFINE_DOM_SOURCE` for details about `DEFINE_DOM_SOURCE` functions.

6.2.6 Hooking `DEFINE_DOM_SPECULAR_REFLECTIVITY` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DOM_SPECULAR_REFLECTIVITY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.8) in ANSYS FLUENT.

i The discrete ordinates (DO) radiation model must be enabled from the Radiation Model dialog box.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.2.8)

Define ⟶ User-Defined ⟶Function Hooks...
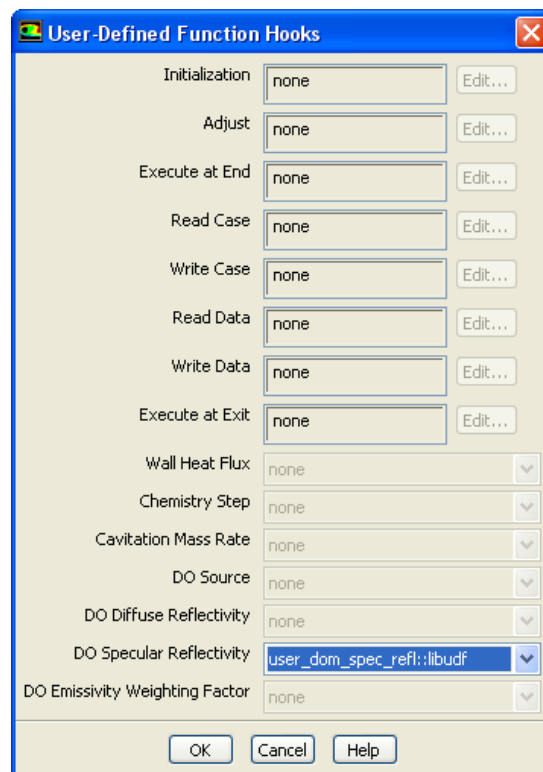
Figure 6.2.8: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., user_dom_spec_refl::libudf) in the DO Specular Reflectivity drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.4: `DEFINE_DOM_DIFFUSE_REFLECTIVITY` for details about `DEFINE_DOM_SPECULAR_REFLECTIVITY` functions.

6.2.7 **Hooking** DEFINE_EMISSIVITY_WEIGHTING_FACTOR **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_EMISSIVITY_WEIGHTING_FACTOR UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.9) in ANSYS FLUENT.

i In the Radiation Model dialog box, Discrete Ordinates (DO) must be selected from the Model list, and a nonzero value must be entered for Number of Bands in the Non-Gray Model group box.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.2.9)
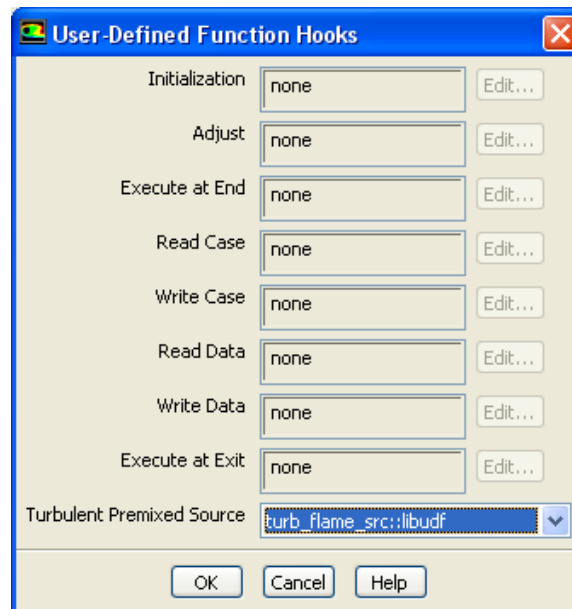
Define ⟶ User-Defined ⟶Function Hooks...
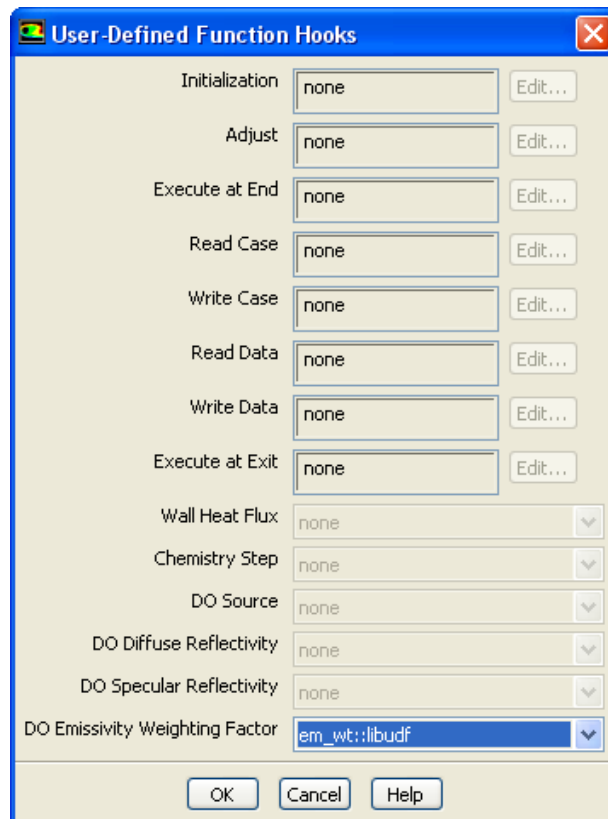
Figure 6.2.9: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., em_wt::libudf) in the DO Emissivity Weighting Factor drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.7: DEFINE_EMISSIVITY_WEIGHTING_FACTOR for details about defining DEFINE_EMISSIVITY_WEIGHTING_FACTOR UDFs.

6.2.8   Hooking DEFINE_GRAY_BAND_ABS_COEFF UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_GRAY_BAND_ABS_COEFF UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Create/Edit Materials dialog box in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first make sure that the Discrete Ordinates (DO) model is selected in the Radiation Model dialog box, with a nonzero Number of Bands in the Non-Gray Model group box. Then open the Materials task page.

⟐ Materials

Select the appropriate material from the Material selection list and click the Create/Edit... button to open the Create/Edit Materials dialog box (Figure 6.2.10).
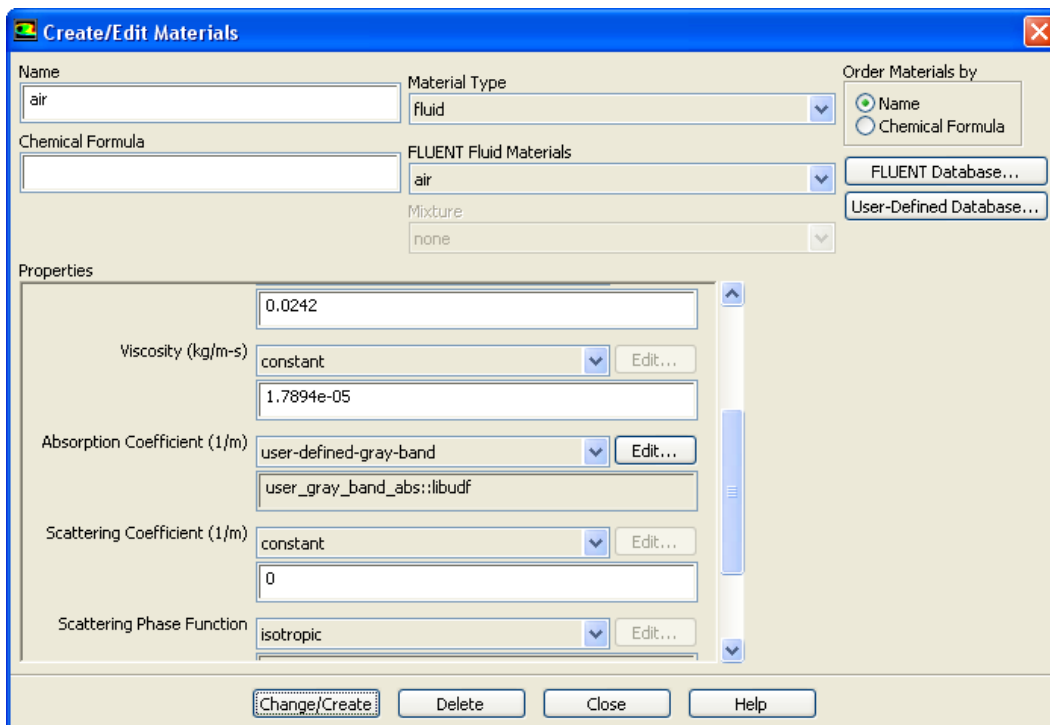
Figure 6.2.10: The Create/Edit Materials Dialog Box

Next, select user-defined-gray-band from the Absorption Coefficient drop-down list in the Create/Edit Materials dialog box. This will open the User-Defined Functions dialog box,

where you must select the name of the function (e.g., **user_gray_band_abs::libudf**) and click OK. Finally, click **Change/Create** in the **Create/Edit Materials** dialog box.

See Section 2.3.8: `DEFINE_GRAY_BAND_ABS_COEFF` for details about `DEFINE_GRAY_BAND_ABS_COEFF` functions.

6.2.9 **Hooking** `DEFINE_HEAT_FLUX` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_HEAT_FLUX` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.2.11) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, open the **User-Defined Function Hooks** dialog box. (Figure 6.2.11)

Define $\longrightarrow$ User-Defined $\longrightarrow$ Function Hooks...
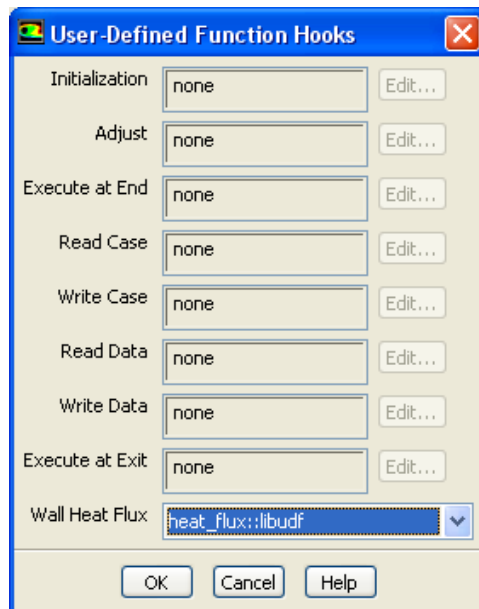
Figure 6.2.11: The **User-Defined Function Hooks** Dialog Box

> ⓘ The **Energy Equation** must be enabled.

Select the function name (e.g., **user_heat_flux::libudf**) in the **Wall Heat Flux** drop-down list in the **User-Defined Function Hooks** dialog box, and click OK.

See Section 2.3.9: `DEFINE_HEAT_FLUX` for details about `DEFINE_HEAT_FLUX` functions.

6.2.10 Hooking `DEFINE_IGNITE_SOURCE` UDFs

After you have compiled (Chapter 5: Compiling UDFs) your `DEFINE_IGNITE_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.12) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first open the General task page.

✧ General

Select Pressure-Based from the Type list, and select Transient from the Time list.

Then, select a turbulence model in the Viscous Model dialog box.

✧ Models ⟶ Viscous Model ⟶ Edit...

Next, set up an appropriate reaction model in the Species Model dialog box.

✧ Models ⟶ Species ⟶ Edit...

Select either the Premixed Combustion or the Partially Premixed Combustion model in the Species Model dialog box and click OK.

Then open the Autoignition Model dialog box.

✧ Models ⟶ Autoignition ⟶ Edit...

Select the Knock Model from the Model list in the Autoignition Model dialog box, and click OK.

Next, open the User-Defined Function Hooks dialog box. (Figure 6.2.12)
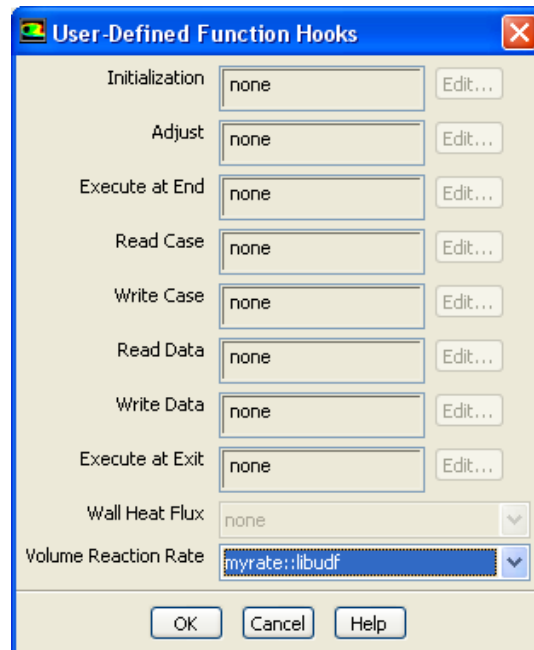
Define ⟶ User-Defined ⟶ Function Hooks...
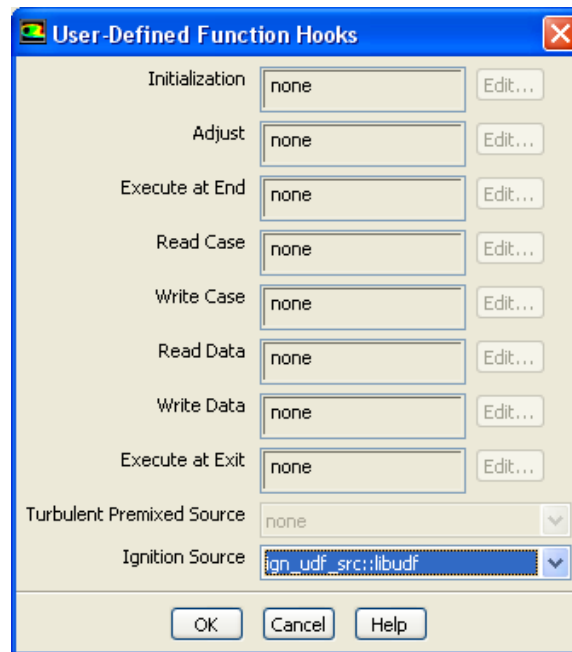
Figure 6.2.12: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., ign_udf_src::libudf) in the Ignition Source drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.10: DEFINE_IGNITE_SOURCE for details about DEFINE_IGNITE_SOURCE functions.

6.2.11   Hooking DEFINE_NET_REACTION_RATE UDFs

After you have compiled (Chapter 5: Compiling UDFs) your DEFINE_NET_REACTION_RATE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.13) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first set up the species transport and combustion models.

⬦ Models ⟶ ≣ Species ⟶ Edit...

Note that net reaction rate UDFs may only be used with the laminar finite-rate model (with stiff chemistry enabled), the EDC model, the PDF Transport model, or the surface chemistry model. Therefore, you must use one of the following groups of settings in the Species Model dialog box:

- To enable the laminar finite-rate model, select Species Transport, enable Volumetric in the Reactions group box, select Laminar Finite-Rate in the Turbulence-Chemistry Interaction group box, and enable Stiff Chemistry Solver in the Options group box.

- To enable the EDC model, select Species Transport, enable Volumetric in the Reactions group box, and select Eddy-Dissipation Concept in the Turbulence-Chemistry Interaction group box.

- To enable the PDF Transport model, select Composition PDF Transport and enable Volumetric in the Reactions group box.

- To enable the surface chemistry model, select Species Transport and enable Volumetric and Wall Surface in the Reactions group box.

Next, open the User-Defined Function Hooks dialog box. (Figure 6.2.13)

Define ⟶ User-Defined ⟶Function Hooks...
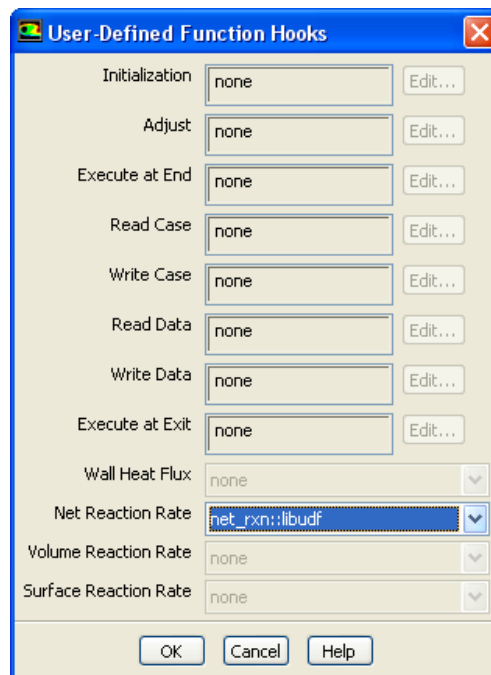
Figure 6.2.13: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., net_rxn::libudf) in the Net Reaction Rate Function drop-down list, and click OK.

See Section 2.3.11: DEFINE_NET_REACTION_RATE for details about DEFINE_NET_REACTION_RATE functions.

6.2.12 Hooking DEFINE_NOX_RATE UDFs

After you have compiled (Chapter 5: Compiling UDFs) your DEFINE_NOX_RATE UDF in ANSYS FLUENT, the function name you supplied in the DEFINE macro argument will become visible and selectable in the NOx Rate drop-down list in the Formation tab of the NOx Model dialog box (Figure 6.2.14).

✧ Models ⟶ ☰ NOx ⟶ Edit...
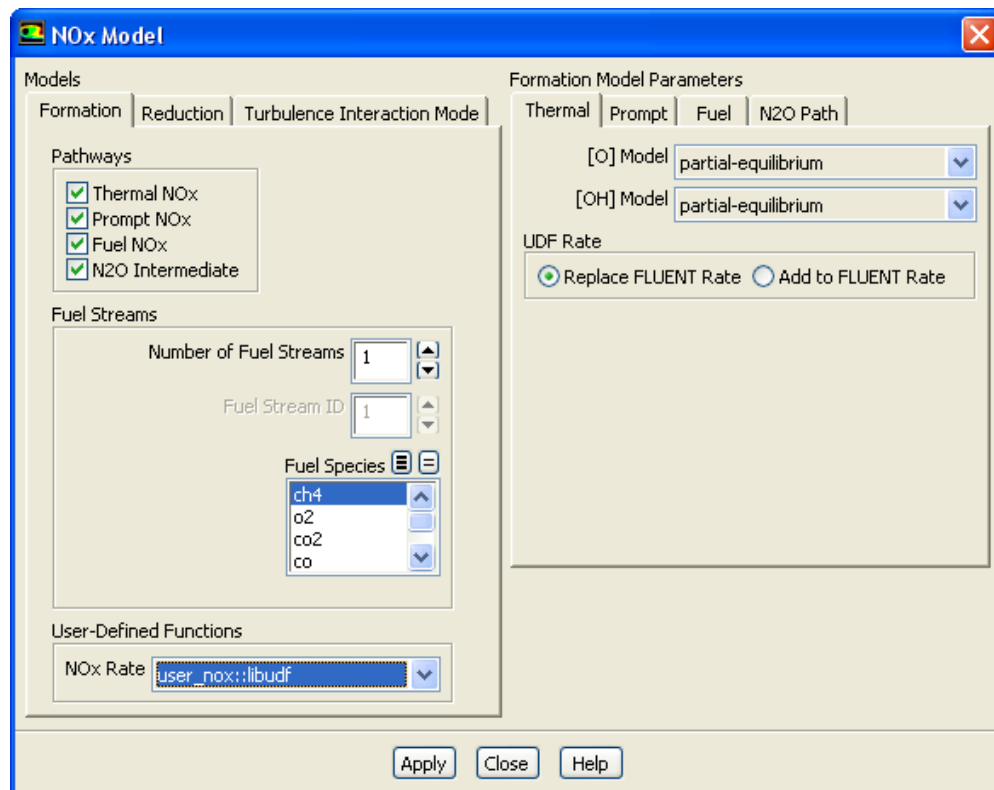
Figure 6.2.14: The NOx Model Dialog Box

Recall that a single UDF is used to define custom rates for the thermal NO_x, prompt NO_x, fuel NO_x, and N_2O NO_x pathways. By default, the custom NO_x rate of your UDF is added to the rate calculated internally by ANSYS FLUENT for each pathway. The UDF rate will be added to the forward rate if it is assigned to the POLLUT_FRATE macro, or the reverse rate if it is assigned to the POLLUT_RRATE macro. If you would rather entirely

replace the internally calculated NO_x rate with your custom rate, click the desired NO_x pathway tab (Thermal, Prompt, Fuel, or N2O Path) under Formation Model Parameters, select Replace FLUENT Rate in the UDF Rate group box for that pathway, and then click Apply. Repeat this process for all of the remaining NO_x pathways.

Unless specifically defined in your NO_x rate UDF, data and parameter settings for each individual NO_x pathway will be derived from the settings in the NOx Model dialog box. Therefore, it is good practice to make the appropriate settings in the NOx Model dialog box, even though you may use a UDF to replace the default rates with user-specified rates. There is no computational penalty for doing this because the default rate calculations will be ignored when Replace FLUENT Rate is selected.

To specify a custom maximum limit ($T_{\max}$) for the integration of the temperature PDF for each cell, you must first select the UDF name (e.g., user_nox::libudf) from the NOx Rate drop-down list, as described previously. Then, click the Turbulence Interaction Mode tab and select either temperature or temperature/species from the PDF Mode drop-down list. Finally, select user-defined from the Tmax Option drop-down list and click Apply.

See Section 2.3.12: `DEFINE_NOX_RATE` for details about defining `DEFINE_NOX_RATE` functions.

6.2.13 Hooking `DEFINE_PR_RATE` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_PR_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.15) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, open the User-Defined Function Hooks dialog box. (Figure 6.2.15)

Define ⟶ User-Defined ⟶Function Hooks...
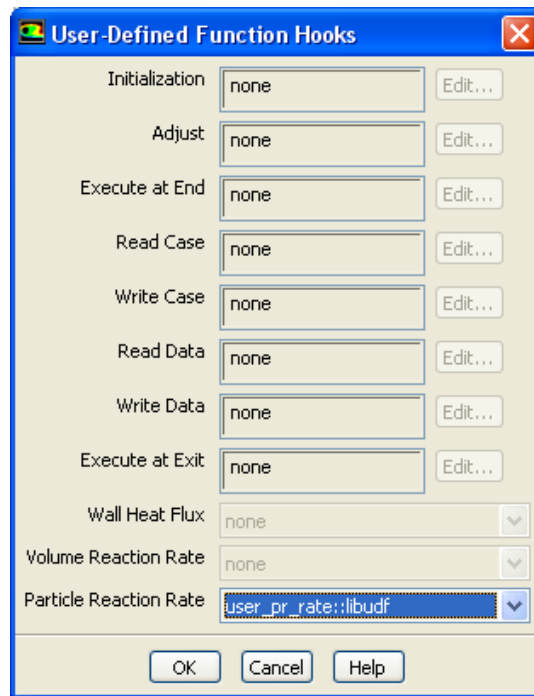
Figure 6.2.15: The User-Defined Function Hooks Dialog Box

ℹ️ You must enable the particle surface reactions option before you can hook the UDF by selecting Volumetric and Particle Surface under Reactions in the Species Model dialog box.

Select the function name (e.g., user_pr_rate::libudf) in the Particle Reaction Rate Function drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.13: DEFINE_PR_RATE for details about defining DEFINE_PR_RATE functions.

6.2.14 Hooking DEFINE_PRANDTL UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_PRANDTL UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Viscous Model dialog box (Figure 6.2.16) in ANSYS FLUENT.

✧ Models ⟶ ≣ Viscous ⟶ Edit...
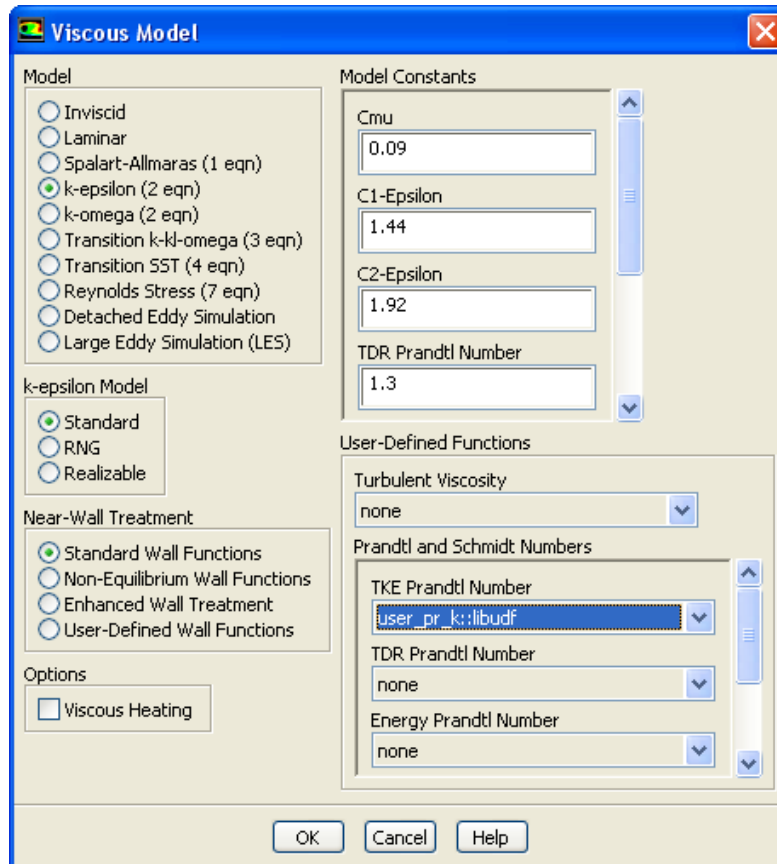
Figure 6.2.16: The Viscous Model Dialog Box

To hook the UDF to ANSYS FLUENT, select the function name (e.g., user_pr_k::libudf) from the TKE Prandtl Number drop-down list under User-Defined Functions in the Viscous Model dialog box, and click OK.

See Section 2.3.14: DEFINE_PRANDTL UDFs for details about DEFINE_PRANDTL functions.

6.2.15   Hooking `DEFINE_PROFILE` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_PROFILE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the appropriate boundary or cell zone condition dialog box in ANSYS FLUENT. To open the boundary or cell zone condition dialog box, select the zone in the Boundary Conditions or Cell Zone Conditions task page and click the Edit... button.

⬥ Boundary Conditions

or

⬥ Cell Zone Conditions

To hook the UDF, select the name of your function in the appropriate drop-down list. For example, if your UDF defines a velocity inlet boundary condition, click the Momentum tab in the Velocity Inlet dialog box (Figure 6.2.17), select the function name (e.g., x_velocity::libudf) from the X Velocity drop-down list, and click OK. Note that the UDF name that is displayed in the drop-down lists is preceded by the word udf (e.g., udf x_velocity::libudf).
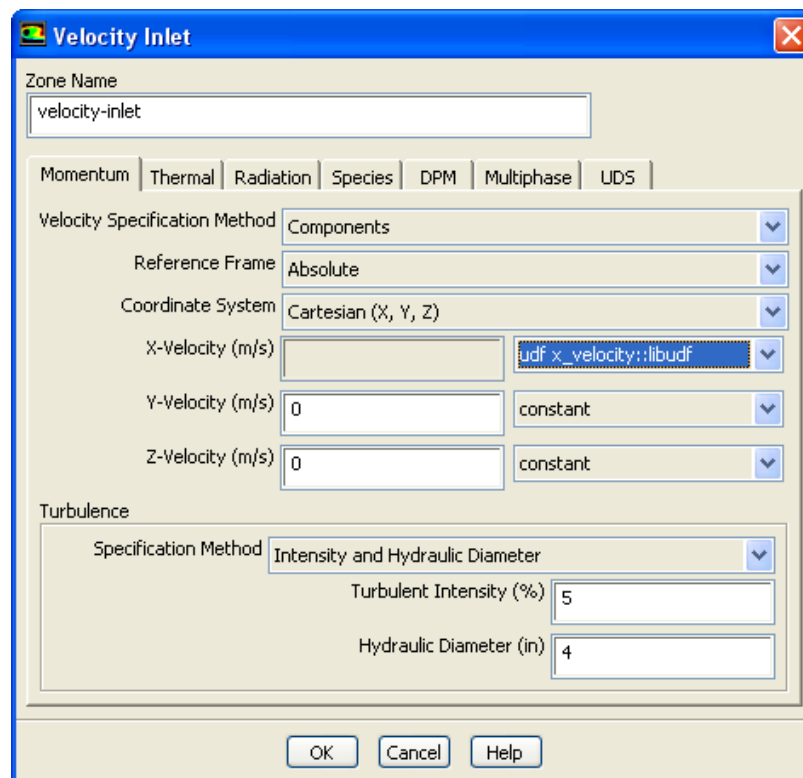
Figure 6.2.17: The Velocity Inlet Dialog Box

If you are using your UDF to specify a fixed value in a cell zone, you will need to turn on the Fixed Values option in the Fluid or Solid dialog box. Then click the Fixed Values tab and select the name of the UDF in the appropriate drop-down list for the value you wish to set.

See Section 2.3.15: DEFINE_PROFILE for details about DEFINE_PROFILE functions.

Hooking Profiles for UDS Equations

For each of the N scalar equations you have specified in your ANSYS FLUENT model using the User-Defined Scalars dialog box you can hook a fixed value UDF for a cell zone (e.g., Fluid or Solid) and a specified value or flux UDF for all wall, inflow, and outflow boundaries.

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_PROFILE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the appropriate boundary or cell zone condition dialog box in ANSYS FLUENT. To open the boundary or cell zone condition dialog box, select the zone in the Boundary Conditions or Cell Zone Conditions task page and click the Edit... button.

✧ Boundary Conditions

or

✧ Cell Zone Conditions

- If you are using your UDF to specify a fixed value in a cell zone, you will need to enable the Fixed Values option in the Fluid or Solid dialog box. Then click the Fixed Values tab (Figure 6.2.18) and select the name of the UDF (e.g., fixed_scalar_0) in the appropriate drop-down list for the value you wish to set.
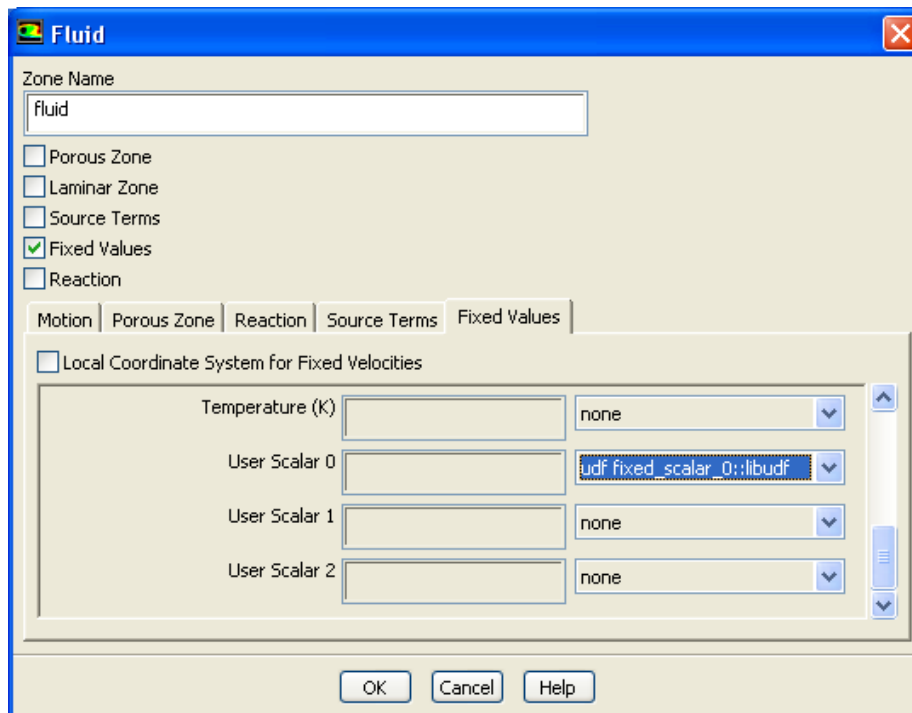
Figure 6.2.18: The Fluid Dialog Box with Fixed Value Inputs for User-Defined Scalars

- If you are using your UDF to define a specific value or flux for a scalar equation in a boundary condition dialog box, you will first need to enter a nonzero number in the User-Defined Scalars text box in the User-Defined Scalars dialog box.

Define $\longrightarrow$ User-Defined $\longrightarrow$ Scalars...

Next, select the UDS tab in the wall, inflow, or outflow boundary dialog box (Figure 6.2.19).
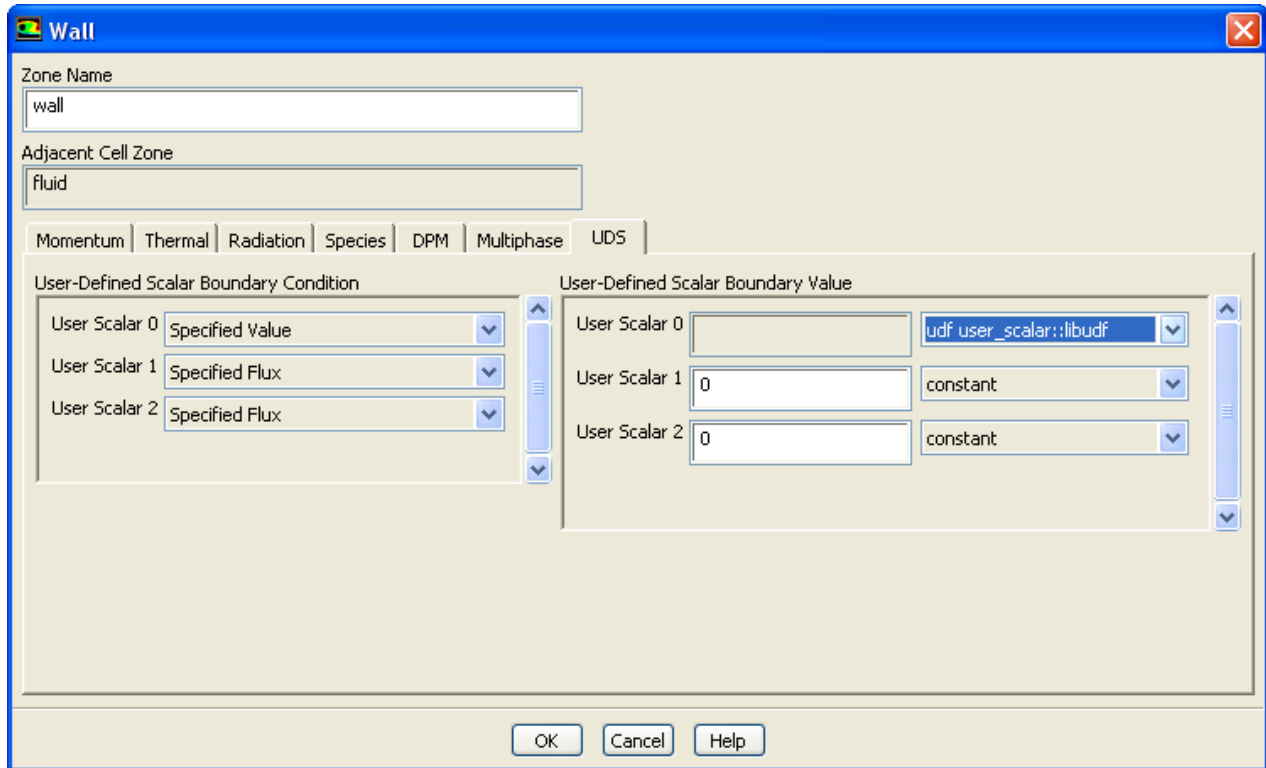
Figure 6.2.19: The Wall Dialog Box with Inputs for User-Defined Scalars

For each UDS (User Scalar 0, User Scalar 1, etc.) specify the boundary condition value as a constant value or a UDF (e.g., user_scalar::libudf) in the User-Defined Scalar Boundary Value group box. If you select Specified Flux in the User-Defined Scalar Boundary Condition group box for a particular UDS, then your input will be the value of the flux at the boundary (i.e., the dot product of the negative of the term in parentheses on the left hand side of Equation 1.3-2 (in the separate Theory Guide) with the vector that is normal to the domain); if you instead select Specified Value, then your input will be the value of the scalar itself at the boundary. In the sample dialog box shown previously, for example, the Specified Value for User Scalar 0 is set to a `user_scalar` UDF.

Note that for interior walls, you will need to select Coupled Boundary if the scalars are to be solved on both sides of a two-sided wall. Note that the Coupled Boundary option will show up only in the drop-down list when all zones is selected for Solution Zones in the User-Defined Scalars dialog box.

i In some cases, you may wish to exclude diffusion of the scalar at the inlet of your domain. You can do this by disabling Inlet Diffusion for the scalar in the User-Defined Scalars dialog box.

See Section 2.3.15: DEFINE_PROFILE for details about DEFINE_PROFILE functions.

6.2.16   Hooking DEFINE_PROPERTY **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your material property UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in ANSYS FLUENT.

There are various dialog boxes in which you can activate a DEFINE_PROPERTY UDF (e.g., Phase Interaction dialog box, Secondary Phase dialog box), and so the method for hooking it will depend on the property being defined. The following is an example of hooking a UDF that defines viscosity.

First, open the Materials task page.

⬦ Materials

Select the appropriate material from the Material selection list and click the Create/Edit... button to open the Create/Edit Materials dialog box (Figure 6.2.20).
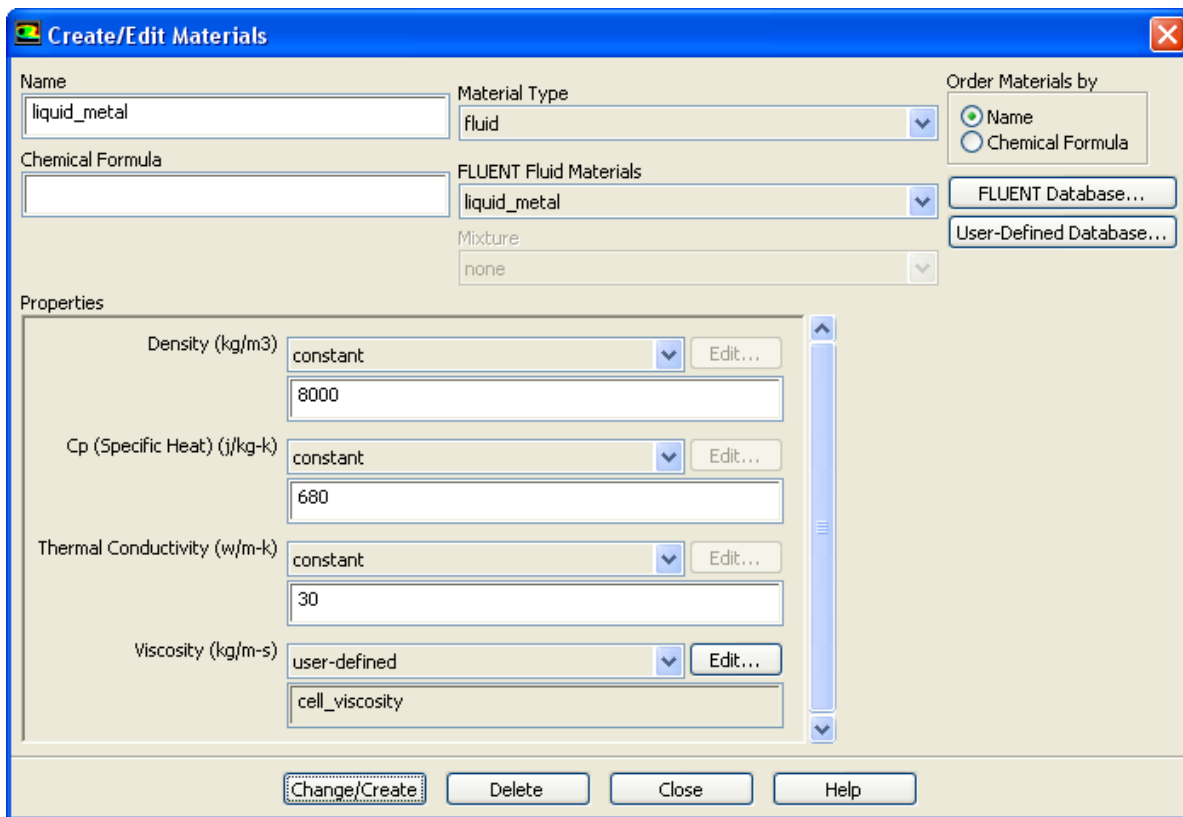
Figure 6.2.20: The Create/ Edit Materials Dialog Box

Next, open the User-Defined Functions dialog box (Figure 6.2.21) by choosing user-defined in the drop-down list for the appropriate property (e.g., Viscosity) in the Create/Edit Materials dialog box. Then select the function name (e.g., cell_viscosity::libudf) from the list of UDFs displayed in the User-Defined Functions dialog box and click OK. The name of the function will subsequently be displayed under the selected property in the Create/Edit Materials dialog box.
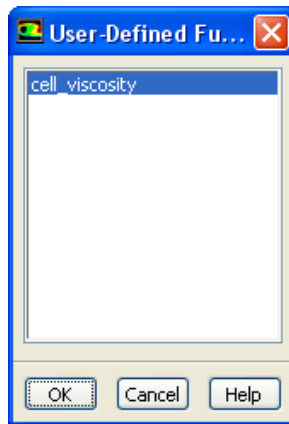
Figure 6.2.21: The User-Defined Functions Dialog Box

i   If you plan to define density using a UDF, note that the solution convergence will become poor as the density variation becomes large. Specifying a compressible law (density as a function of pressure) or multiphase behavior (spatially varying density) may lead to divergence. It is recommended that you restrict the use of UDFs for density to weakly compressible flows with mild density variations.

See Section 2.3.16: DEFINE_PROPERTY UDFs for details about DEFINE_PROPERTY functions.

6.2.17   **Hooking** DEFINE_SCAT_PHASE_FUNC **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_SCAT_PHASE_FUNC UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Functions dialog box in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first make sure that the Discrete Ordinates (DO) model is selected in the Radiation Model dialog box. Then open the Materials task page.

⇳   Materials

Select the appropriate material from the Material selection list and click the Create/Edit... button to open the Create/Edit Materials dialog box (Figure 6.2.22).
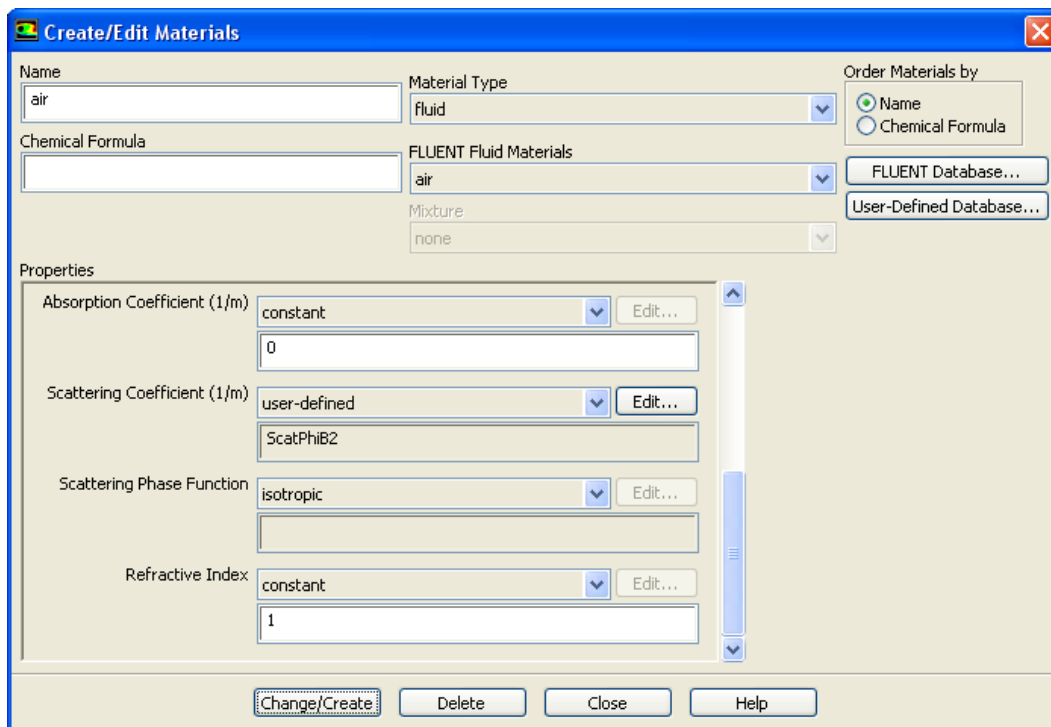
Figure 6.2.22: The Create/Edit Materials Dialog Box

Open the User-Defined Functions dialog box (Figure 6.2.23) from the Create/Edit Material dialog box by selecting user-defined in the drop-down list for the Scattering Phase Function property. Then, select the function name (e.g., ScatPhiB2) from the list of UDFs displayed in the User-Defined Functions dialog box, and click OK. The name of the function will subsequently be displayed under the Scattering Phase Function property in the Create/Edit Materials dialog box.
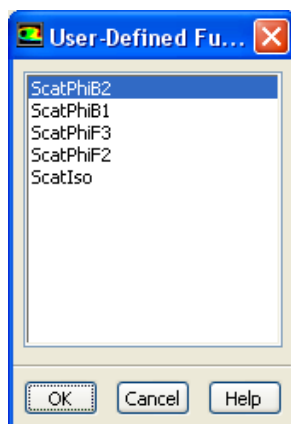
Figure 6.2.23: The User-Defined Functions Dialog Box

See Section 2.3.17: DEFINE_SCAT_PHASE_FUNC for details about DEFINE_SCAT_PHASE_FUNC functions.

6.2.18   **Hooking** DEFINE_SOLAR_INTENSITY **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_SOLAR_INTENSITY UDF, the name of the function you supplied in the argument of the DEFINE macro will become visible and selectable in the User-Defined Functions dialog box in ANSYS FLUENT.

To hook the UDF, first open the Radiation Model dialog box (Figure 6.2.24).

⬦ Models ⟶ ▤ Radiation ⟶ Edit...
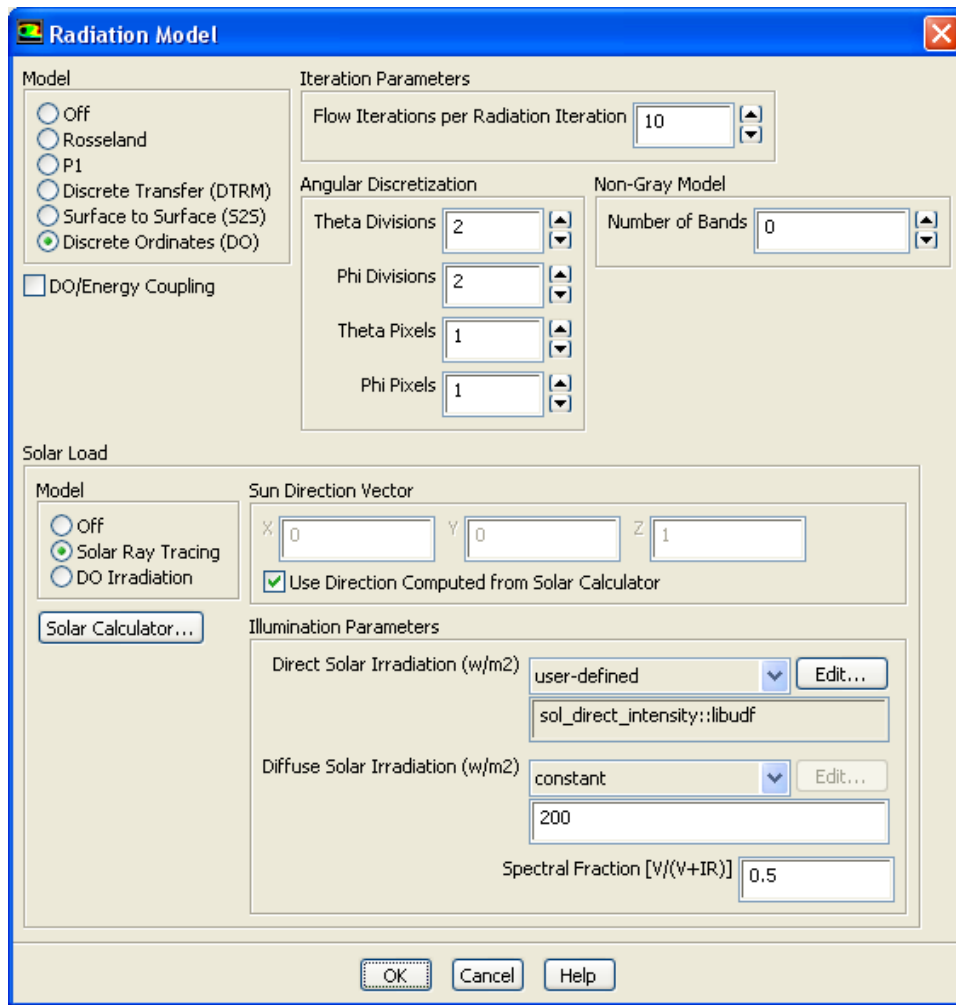
Figure 6.2.24: The Radiation Model Dialog Box

Select Discrete Ordinates (DO) from the Model list, and select Solar Ray Tracing in the Solar Load group box. In the Illumination Parameters group box that appears, select user-defined from the Direct Solar Irradiation or Diffuse Solar Irradiation drop-down list to open the User-Defined Functions dialog box (Figure 6.2.25).
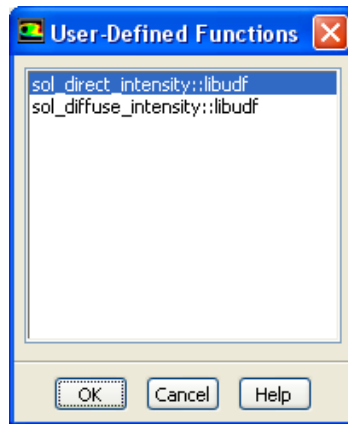
Figure 6.2.25: The User-Defined Functions Dialog Box

Select the function name (e.g., sol_direct_intensity::libudf) from the list of UDFs displayed in the User-Defined Functions dialog box and click OK. The name of the function will subsequently be displayed under the selected property (e.g., Direct Solar Irradiation) in the Radiation Model dialog box (Figure 6.2.24).

See Section 2.3.18: `DEFINE_SOLAR_INTENSITY` for details about `DEFINE_SOLAR_INTENSITY` functions.

6.2.19 Hooking `DEFINE_SOURCE` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in a source term dialog box (e.g., the Mass sources dialog box) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, you will first need to open the Cell Zone Conditions task page.

↔ Cell Zone Conditions

Select the appropriate zone in the Zone selection list of the Cell Zone Conditions task page and click the Edit... button to open the cell zone condition dialog box (e.g., the Fluid dialog box, as shown in Figure 6.2.26).
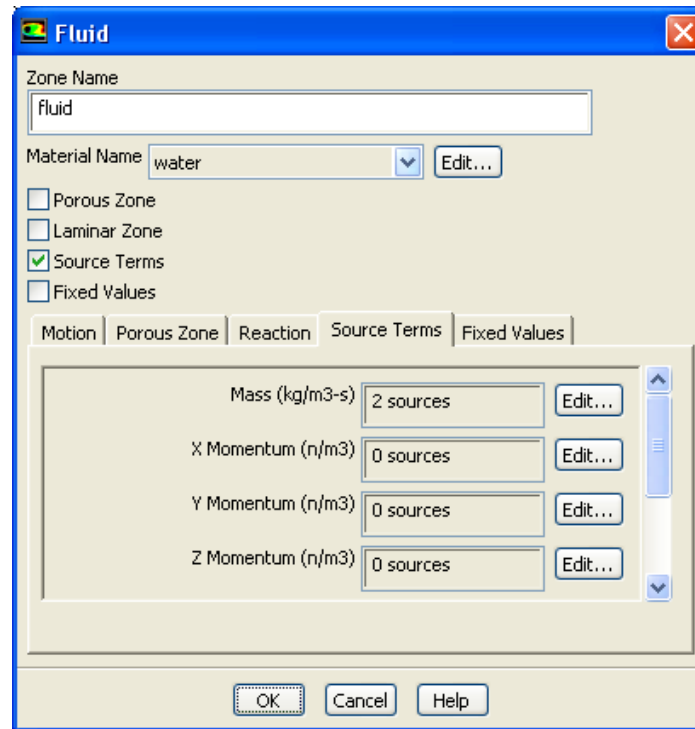
Figure 6.2.26: The Fluid Dialog Box

Next, enable the Source Terms option in the cell zone condition dialog box and click the Source Terms tab. This will display the source term parameters (mass, momentum, etc.) in the scrollable window. Click the Edit... button next to the source term (e.g., Mass) you wish to customize, in order to open the appropriate source term dialog box (e.g., the Mass sources dialog box, as shown in Figure 6.2.27).

Specify the number of terms you wish to model by setting the Number of Mass Sources text-entry box (e.g., 2) and then select the function name (e.g., usr_mass_src1::libudf and usr_mass_src2::libudf) from the appropriate drop-down list. (Note that the UDF name that is displayed in the drop-down lists is preceded by the word udf.) Click OK in the Mass sources dialog box to accept the new boundary condition. The source term text box in the cell zone condition dialog box will display the number of sources (e.g., 2 sources). Click OK to close the cell zone condition dialog box and fix the new mass source terms for the solution calculation.

Repeat this step for all of the source terms you wish to customize using a UDF.

See Section 2.3.19: DEFINE_SOURCE for details about DEFINE_SOURCE functions.
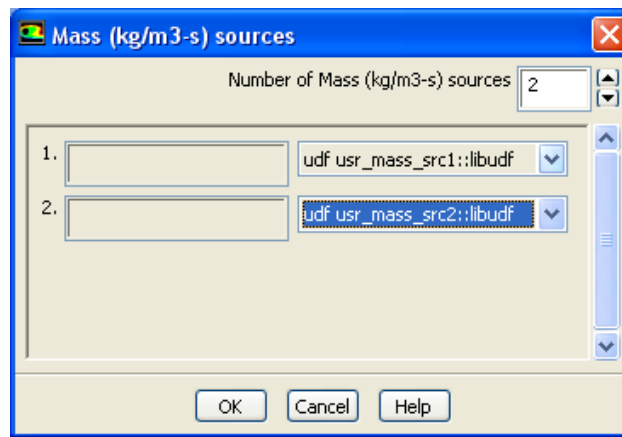
Figure 6.2.27: The **Mass sources** Dialog Box

6.2.20   **Hooking** `DEFINE_SOX_RATE` **UDFs**

After you have compiled (Chapter 5: Compiling UDFs) your `DEFINE_SOX_RATE` UDF in
ANSYS FLUENT, the function name you supplied in the `DEFINE` macro argument (e.g.,
user_sox::libudf) will become visible and selectable for the SOx Rate drop-down list in the
SOx Model dialog box (Figure 6.2.28).

⬦ Models ⟶ ☰ SOx ⟶ Edit...

By default, the custom SO_x rate of your UDF is added to the rate calculated internally
by ANSYS FLUENT. The UDF rate will be added to the forward rate if it is assigned to
the `POLLUT_FRATE` macro, or the reverse rate if it is assigned to the `POLLUT_RRATE` macro.
If you would rather entirely replace the internally calculated SO_x rate with your custom
rate, select Replace FLUENT Rate in the UDF Rate group box and click Apply.

Unless specifically defined in your SO_x rate UDF, data and parameter settings will be
derived from the settings in the SOx Model dialog box. Therefore, it is good practice to
make the appropriate settings in the SOx Model dialog box, even though you may use
a UDF to replace the default rates with user-specified rates. There is no computational
penalty for doing this because the default rate calculations will be ignored when Replace
FLUENT Rate is selected.

To specify a custom maximum limit (T_{max}) for the integration of the temperature PDF
for each cell, you must first select the UDF name (e.g., user_sox::libudf) from the SOx
Rate drop-down list, as described previously. Then, select either temperature or tempera-
ture/species from the PDF Mode drop-down list in the Turbulence Interaction Mode group
box. Finally, select user-defined from the Tmax Option drop-down list and click Apply.

See Section 2.3.20: `DEFINE_SOX_RATE` for details about defining `DEFINE_SOX_RATE` func-
tions.
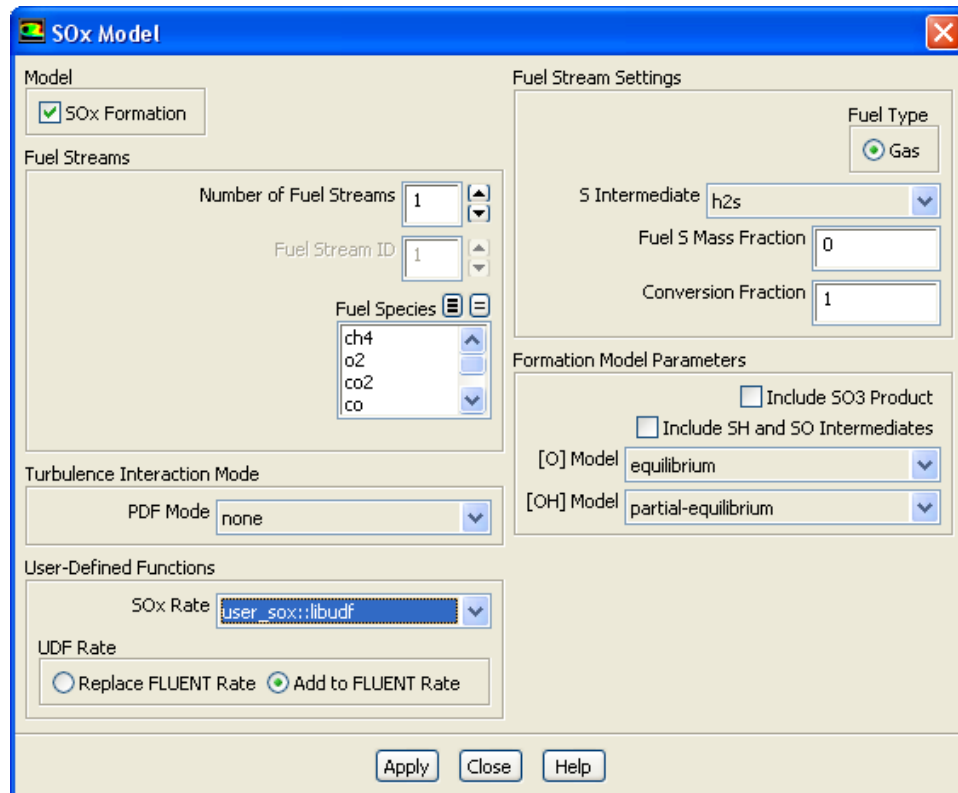
Figure 6.2.28: The SOx Model Dialog Box

6.2.21 **Hooking** DEFINE_SPECIFIC_HEAT **UDFs**

After you have compiled your DEFINE_SPECIFIC_HEAT UDF (as described in Chapter 5: Compiling UDFs), the name of the function you supplied as a DEFINE macro argument will become visible and selectable in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, you will first need to open the Materials task page.

⟡ Materials

Select the appropriate material from the Material selection list and click the Create/Edit... button to open the Create/Edit Materials dialog box (Figure 6.2.29).
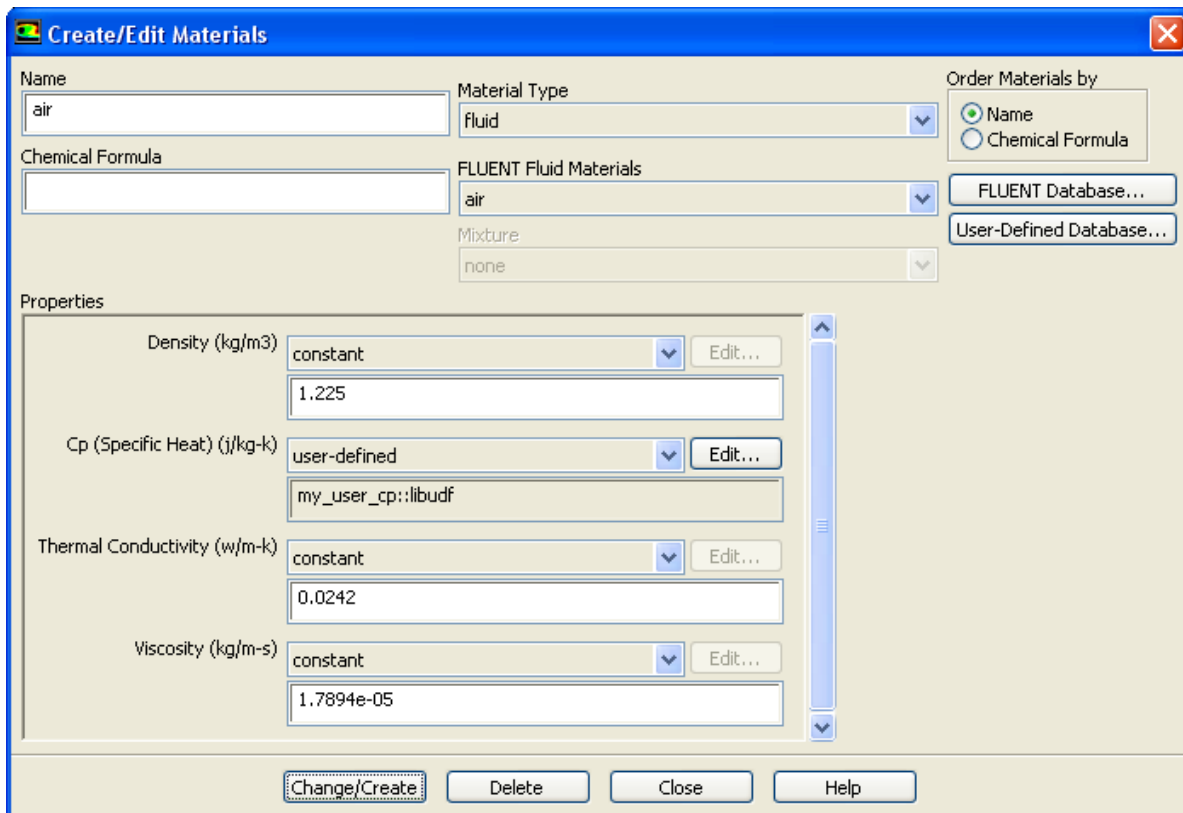
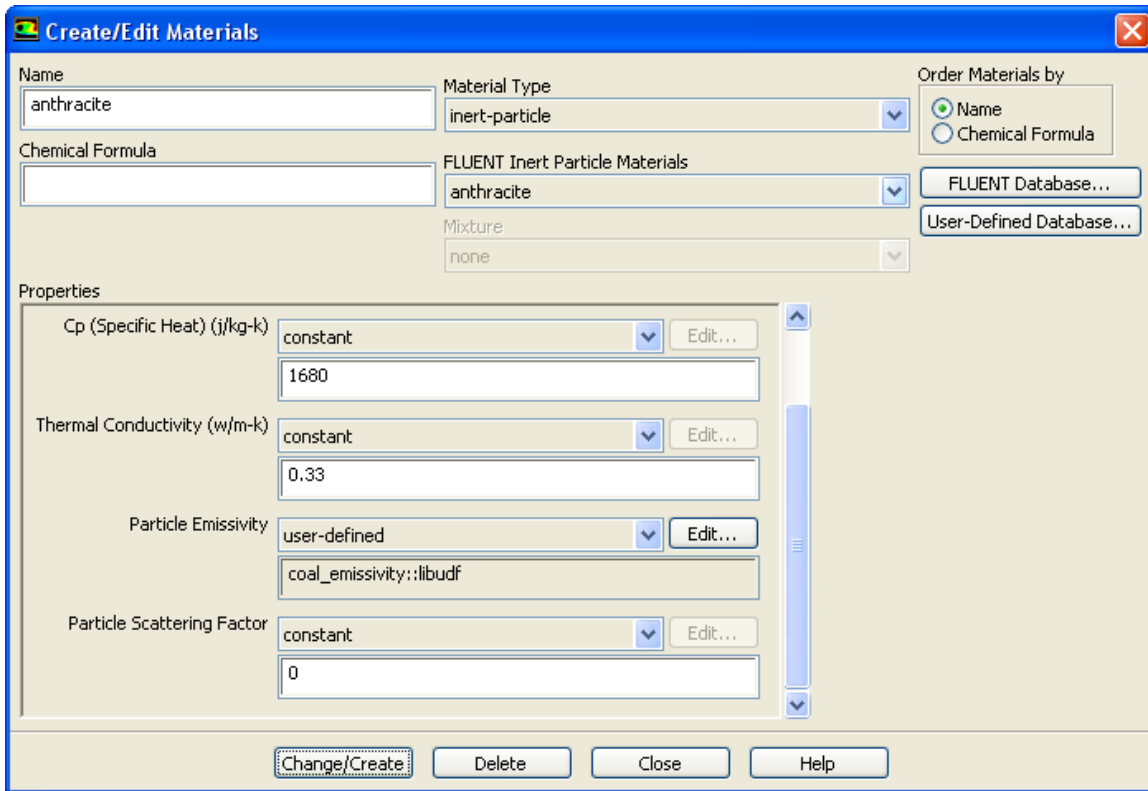Figure 6.2.29: The Create/Edit Materials Dialog Box

Next, select user-defined from the drop-down list for Cp to open the User-Defined Functions dialog box (Figure 6.2.30). Select the name you defined in the UDF (e.g., my_user_cp::libudf) and click OK. The name of the function will subsequently be displayed under the Cp property in the Create/Edit Materials dialog box.

See Section 2.3.21: DEFINE_SPECIFIC_HEAT for details about defining DEFINE_SPECIFIC_HEAT UDFs.
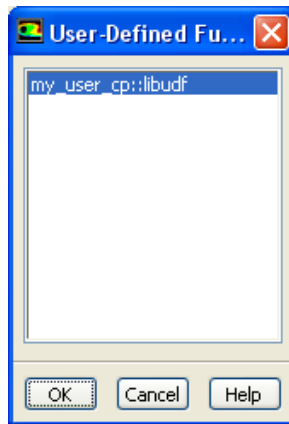
Figure 6.2.30: The User-Defined Functions Dialog Box

6.2.22   Hooking DEFINE_SR_RATE UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_SR_RATE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.31) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first set up an appropriate reaction model in the Species Model dialog box.

⬦ Models ⟶ Species ⟶ Edit...

Select Species Transport from the Model list in the Species Model dialog box, and enable the Volumetric and Wall Surface options in the Reactions group box. Make sure that Stiff Chemistry Solver is disabled in the Options group box, and click OK.

Next, open the User-Defined Function Hooks dialog box. (Figure 6.2.31)

Define ⟶ User-Defined ⟶ Function Hooks...
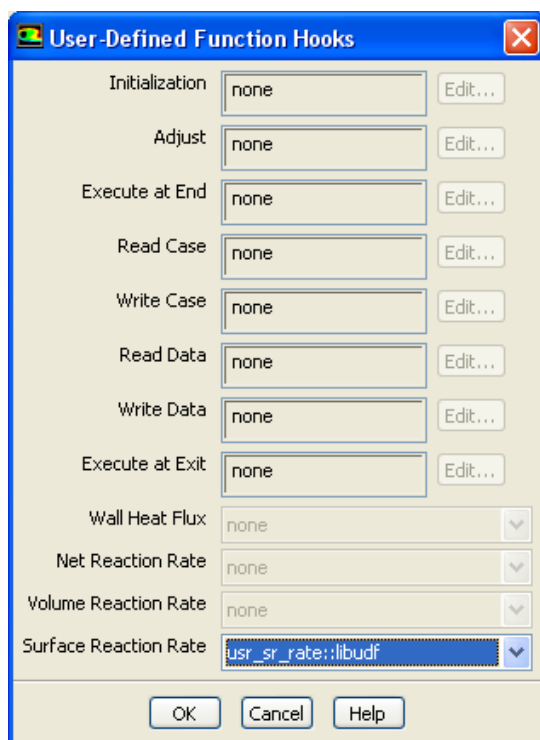
Figure 6.2.31: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., user_sr_rate::libudf) in the Surface Reaction Rate Function drop-down list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.22: DEFINE_SR_RATE for details about DEFINE_SR_RATE functions.

6.2.23 Hooking `DEFINE_TRANS` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_TRANS` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box in ANSYS FLUENT. To hook the UDF, select **Transition SST** from the **Model** list in the **Viscous Model** dialog box (Figure 6.2.32).

⟐ Models ⟶ ☰ Viscous ⟶ Edit...
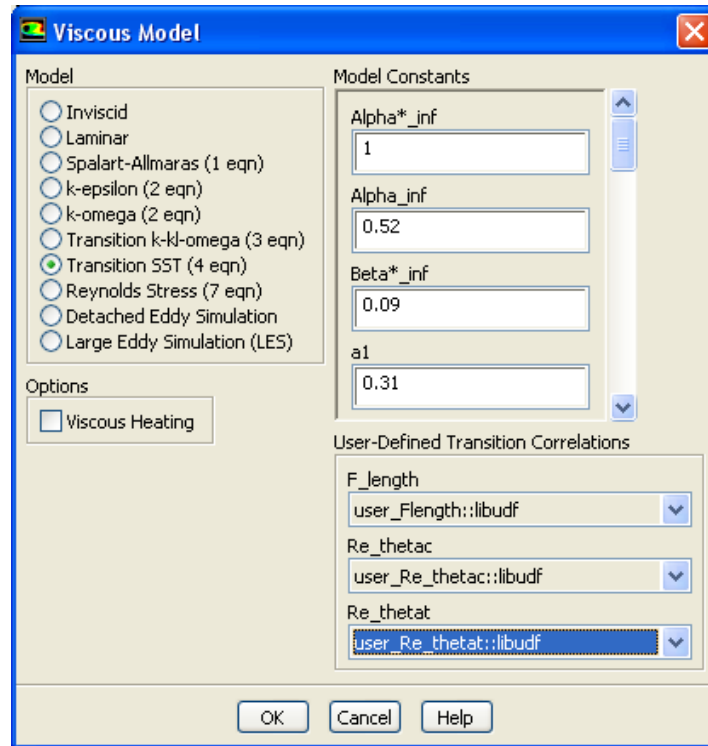
Figure 6.2.32: The **Viscous Model** Dialog Box

Next, select the function name (e.g., user_Flength::libudf) from a drop-down list in the **User-Defined Transition Correlations** group box (e.g., **Flength**), and click **OK**.

See Section 2.3.23: `DEFINE_TRANS` UDFs for details about `DEFINE_TRANS` functions.

6.2.24 **Hooking** DEFINE_TURB_PREMIX_SOURCE **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_TURB_PREMIX_SOURCE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.2.33) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, open the **User-Defined Function Hooks** dialog box. (Figure 6.2.33)

$\boxed{\text{Define}} \longrightarrow \boxed{\text{User-Defined}} \longrightarrow \text{Function Hooks...}$

Figure 6.2.33: The User-Defined Function Hooks Dialog Box

$\boxed{i}$ You must have a premixed combustion model enabled in the Species Model dialog box.

Select the function name (e.g., turb_flame_src::libudf) in the **Turbulent Premixed Source Function** drop-down list in the **User-Defined Function Hooks** dialog box, and click OK.

See Section 2.3.24: DEFINE_TURB_PREMIX_SOURCE for details about DEFINE_TURB_PREMIX_SOURCE functions.

6.2.25 Hooking `DEFINE_TURB_SCHMIDT` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_TURB_SCHMIDT` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Viscous Model dialog box in ANSYS FLUENT. To hook the UDF, first open the Viscous Model dialog box (Figure 6.2.34) and set up a turbulence model.

i   If you select k-epsilon from the Model list, you must not select RNG from the k-epsilon Model list.

⬦ Models ⟶ ☰ Viscous ⟶ Edit...
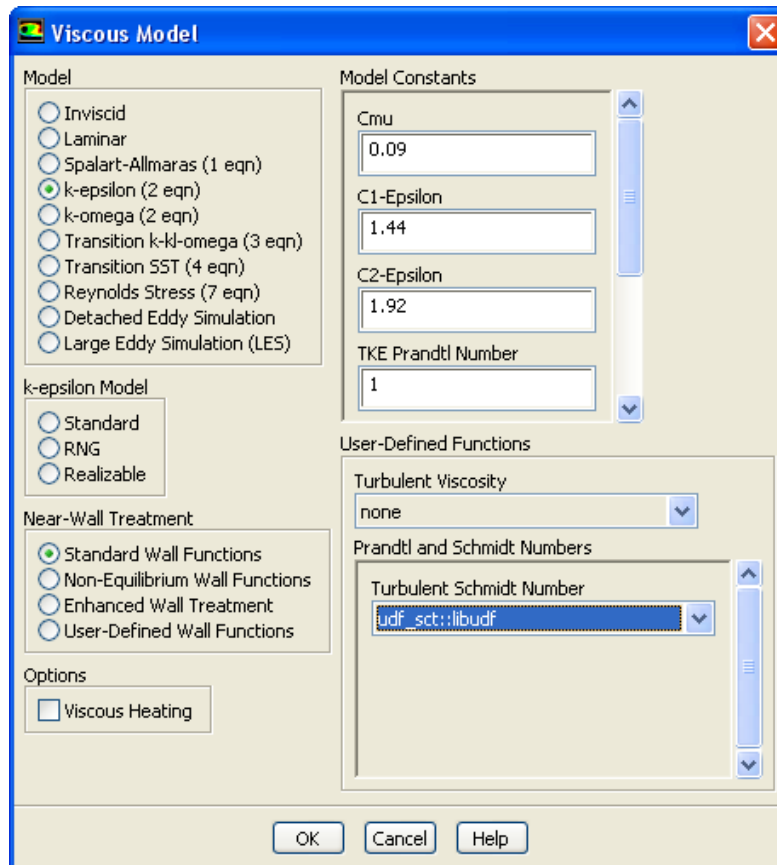
Figure 6.2.34: The Viscous Model Dialog Box

Next, select the function name (e.g., udf_sct::libudf) from the Turbulent Schmidt Number drop-down list under User-Defined Functions in the Viscous Model dialog box, and click OK.

> *i* The Species Transport model must be selected in the Species Model dialog
> box for the **Turbulent Schmidt Number** drop-down list to be visible in the
> Viscous Model dialog box.

See Section 2.3.25: DEFINE_TURB_SCHMIDT UDF for details about DEFINE_TURB_SCHMIDT
functions.

6.2.26   Hooking DEFINE_TURBULENT_VISCOSITY UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Com-
piling UDFs) your DEFINE_TURBULENT_VISCOSITY UDF, the name of the function you
supplied as a DEFINE macro argument will become visible and selectable in the Viscous
Model dialog box (Figure 6.2.35) in ANSYS FLUENT.

↻ Models ⟶ ☰ Viscous ⟶ Edit...
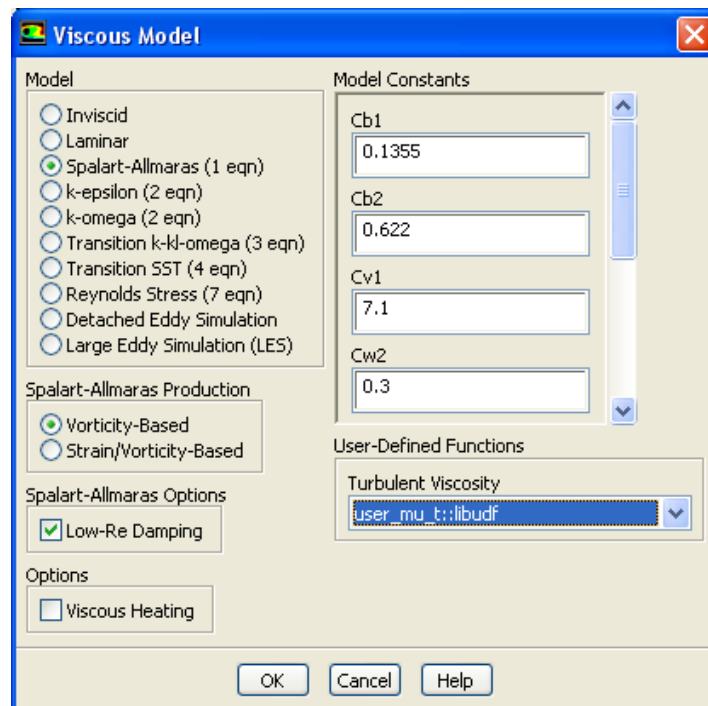
Figure 6.2.35: The Viscous Model Dialog Box

To hook the UDF to ANSYS FLUENT, select the function name (e.g., user_mu_t::libudf)
from the Turbulence Viscosity drop-down list under User-Defined Functions in the Viscous
Model dialog box, and click OK.

See   Section   2.3.26:   DEFINE_TURBULENT_VISCOSITY   for   details   about
DEFINE_TURBULENT_VISCOSITY functions.

6.2.27   Hooking `DEFINE_VR_RATE` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_VR_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.2.36) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first set up an appropriate reaction model in the Species Model dialog box.

✧ Models ⟶ ☰ Species ⟶ Edit...

Select Species Transport from the Model list in the Species Model dialog box, and enable the Volumetric option in the Reactions group box. Make sure that Stiff Chemistry Solver is disabled in the Options group box, and click OK.

Next, open the User-Defined Function Hooks dialog box. (Figure 6.2.36)

Define ⟶ User-Defined ⟶Function Hooks...

Figure 6.2.36: The User-Defined Function Hooks Dialog Box

Select the function name (e.g., myrate::libudf) in the Volume Reaction Rate Function dropdown list in the User-Defined Function Hooks dialog box, and click OK.

See Section 2.3.27: `DEFINE_VR_RATE` for details about `DEFINE_VR_RATE` functions.

6.2.28   Hooking DEFINE_WALL_FUNCTIONS UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_WALL_FUNCTIONS UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Viscous Model dialog box (Figure 6.2.37) in ANSYS FLUENT.

✧ Models ⟶ ☰ Viscous ⟶ Edit...
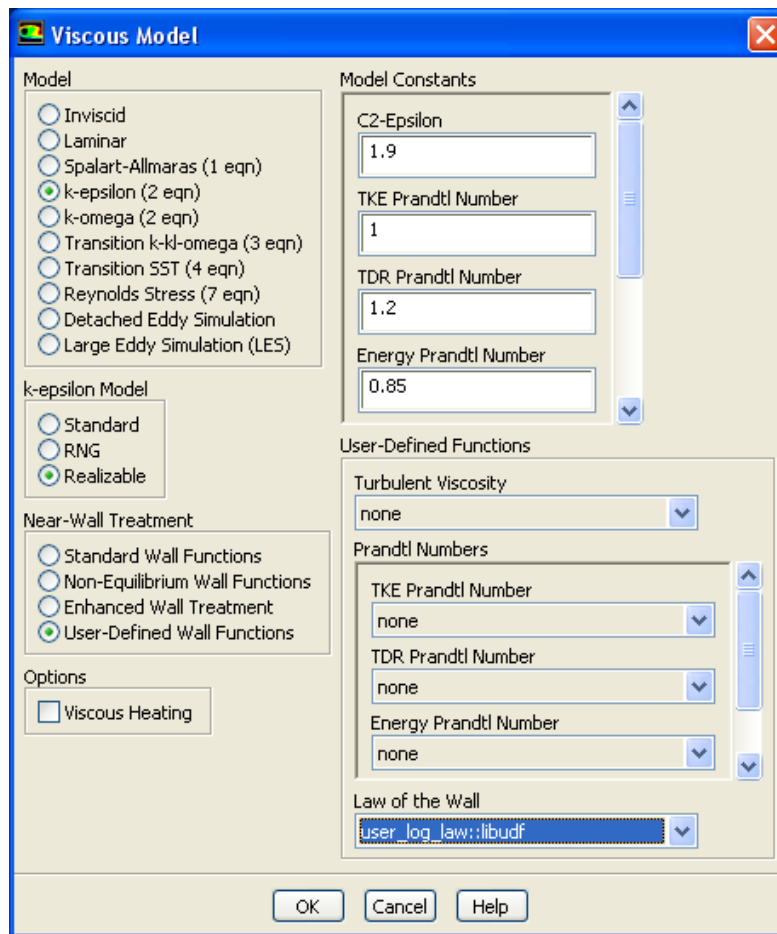
Figure 6.2.37: The Viscous Model Dialog Box

To hook the UDF, select k-epsilon from the Model list in the Viscous Model dialog box, and select User-Defined Wall Functions from the Near-Wall Treatment list. Then, select the function name (e.g., user_log_law::libudf) from the Law of the Wall drop-down list, and click OK.

See Section 2.3.28: DEFINE_WALL_FUNCTIONS for details about defining DEFINE_WALL_FUNCTIONS functions in ANSYS FLUENT.

6.3 Hooking Multiphase UDFs

This section contains methods for hooking UDFs to ANSYS FLUENT that have been defined using DEFINE macros (described in Section 2.4: Multiphase DEFINE Macros), and interpreted or compiled using methods (described in Chapters 4 or 5), respectively.

6.3.1 Hooking DEFINE_CAVITATION_RATE UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_CAVITATION_RATE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Function Hooks dialog box (Figure 6.3.2) in ANSYS FLUENT. Note that cavitation rate UDFs can be applied only to the mixture multiphase model.

To hook the UDF to ANSYS FLUENT, you will first need to open the Multiphase Model dialog box.

✧ Models ⟶ Multiphase ⟶ Edit...

Enable the Mixture model in the Multiphase Model dialog box and click OK.

Enter the solve/set/expert text command in the console and respond yes to use Singhal-et-al cavitation model?. Then open the Phase Interaction dialog box.

✧ Phases ⟶ Interaction...
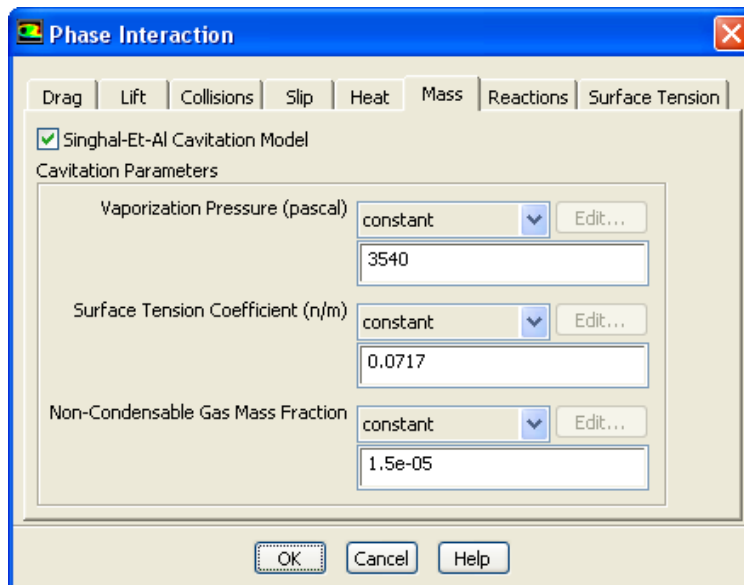
Figure 6.3.1: The Phase Interaction Dialog Box

Enable Singhal-Et-Al Cavitation Model in the Mass tab of the Phase Interaction dialog box (Figure 6.3.1), and click OK.

Next, open the User-Defined Function Hooks dialog box. (Figure 6.3.2)

Define ⟶ User-Defined ⟶Function Hooks...
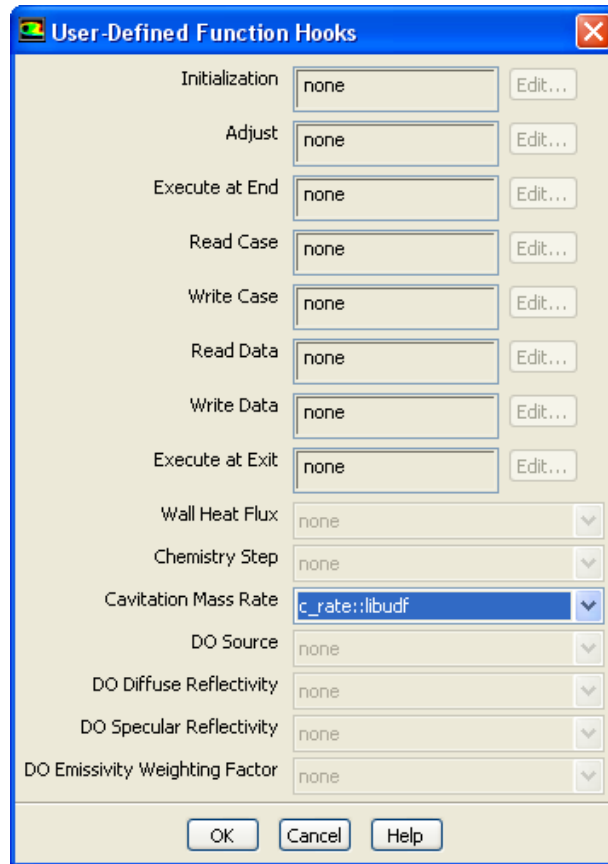
Figure 6.3.2: The User-Defined Function Hooks Dialog Box

To hook the UDF to ANSYS FLUENT, select the function name (e.g., c_rate::libudf) in the Cavitation Mass Rate drop-down list (Figure 6.3.2), and click OK.

See Section 2.4.1: DEFINE_CAVITATION_RATE for details about DEFINE_CAVITATION_RATE functions.

6.3.2 Hooking DEFINE_EXCHANGE_PROPERTY UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_EXCHANGE_RATE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in ANSYS FLUENT.

To hook an exchange property UDF to ANSYS FLUENT, first open the Multiphase Model dialog box.

✧ Models ⟶ 🗎 Multiphase ⟶ Edit...

Customized mass transfer UDFs can be applied to VOF, Mixture, and Eulerian multiphase models. Drag coefficient UDFs can be applied to Mixture and Eulerian models, while heat transfer and lift coefficient UDFs can be applied only to the Eulerian model. Select the appropriate model from the Model list in the Multiphase Model dialog box and click OK.

i Make sure that you enable Slip Velocity in the Mixture Parameters group box in the Multiphase Model dialog box in order to display the drag coefficient for the Mixture model.

Next, open the Phase Interaction dialog box (Figure 6.3.3).

✧ Phases ⟶ Interaction...
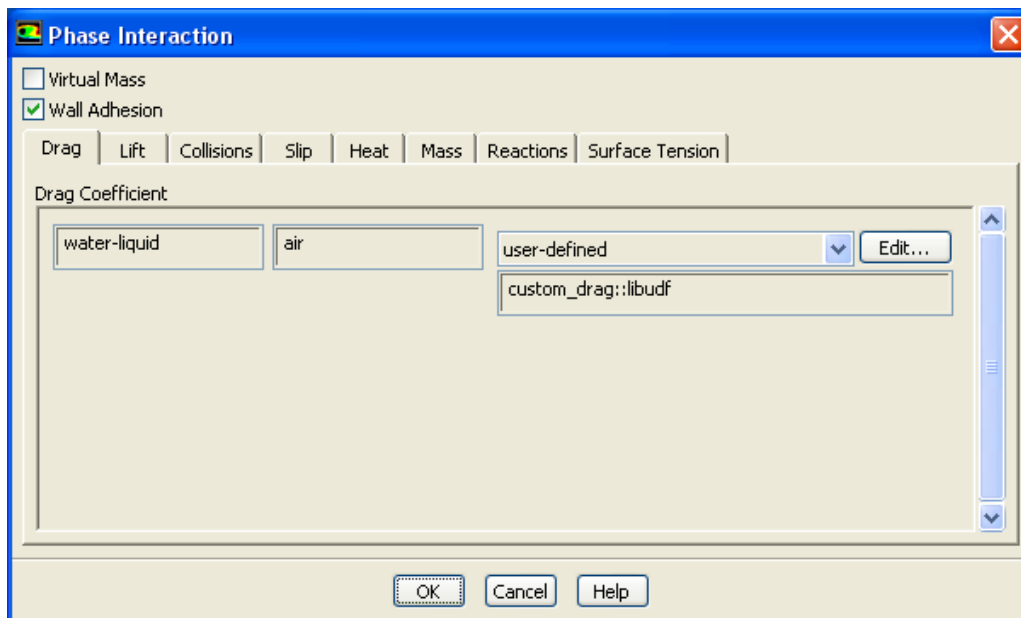
Figure 6.3.3: The Phase Interaction Dialog Box

Click the appropriate tab (e.g., Drag) in the Phase Interaction dialog box, and select user-defined from the drop-down list for the corresponding exchange property (e.g., Drag Coefficient) that you desire. This will open the User-Defined Functions dialog box.
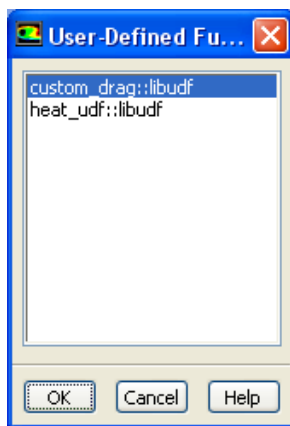
Figure 6.3.4: The User-Defined Functions Dialog Box

Select the function name (e.g., custom_drag::libudf) from the list of UDFs displayed in the User-Defined Functions dialog box (Figure 6.3.4) and click OK. The function name will then be displayed in a text box under the exchange property in the Phase Interaction dialog box. Click OK to close the Phase Interaction dialog box.

See Section 2.4.2: DEFINE_EXCHANGE_PROPERTY for details about DEFINE_EXCHANGE_PROPERTY functions.

6.3.3 Hooking `DEFINE_HET_RXN_RATE` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_HET_RXN_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in ANSYS FLUENT.

To hook the UDF, first make a selection from the Model list of the Multiphase Model dialog box.

✧ Models ⟶ ≣ Multiphase ⟶ Edit...

Next, select Species Transport from the Model list of the Species Model dialog box.

✧ Models ⟶ ≣ Species ⟶ Edit...

Then open the Phase Interaction dialog box (Figure 6.3.5).

✧ Phases ⟶ Interaction...
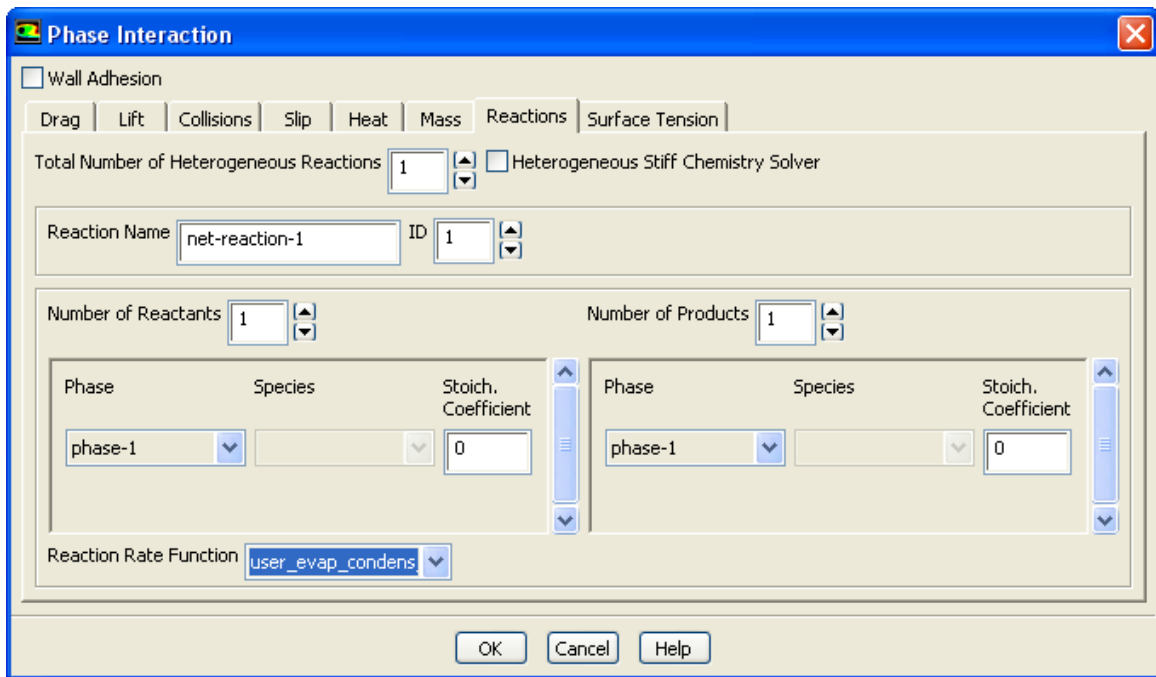
Figure 6.3.5: The Phase Interaction Dialog Box

Click the Reactions tab and enter a nonzero number in the Total Number of Heterogeneous Reactions text box. Select the function name (e.g., user_evap_condens_react::libudf) from the Reaction Rate Function drop-down list and click OK.

See Section 2.4.3: `DEFINE_HET_RXN_RATE` for details about writing `DEFINE_HET_RXN_RATE` functions.

6.3.4   **Hooking** DEFINE_MASS_TRANSFER **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_MASS_TRANSFER UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Phase Interaction dialog box (Figure 6.3.6).

To hook the UDF to ANSYS FLUENT, first make a selection from the Model list of the Multiphase Model dialog box.

⬦ Models ⟶ ≣ Multiphase ⟶ Edit...

Next, open the Phase Interaction dialog box (Figure 6.3.6).

⬦ Phases ⟶ Interaction...
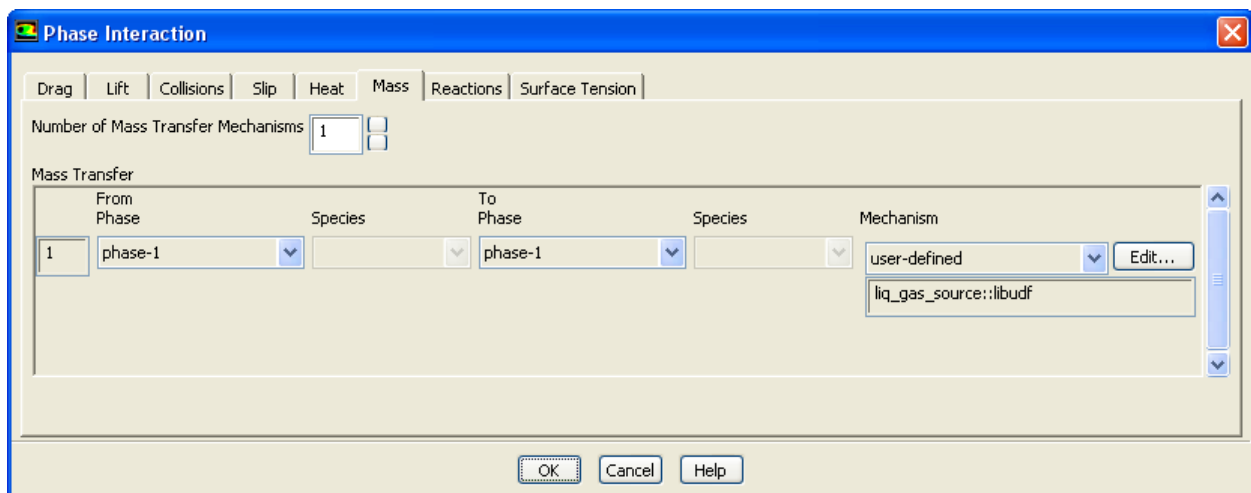
Figure 6.3.6: The Phase Interaction Dialog Box

Click the Mass tab and then specify the Number of Mass Transfer Mechanisms greater than 0. The Mechanism drop-down list will appear. Next, select user-defined from the Mechanism drop-down list to open the User-Defined Functions dialog box (Figure 6.3.7).

Figure 6.3.7: The User-Defined Functions Dialog Box

Select the function name (e.g., liq_gas_source::udf) and click OK. The UDF name will appear in the text box below the Mechanism drop-down list in the Phase Interaction dialog box. Click OK to close the Phase Interaction dialog box.

See Section 2.4.4: DEFINE_MASS_TRANSFER for details about writing DEFINE_MASS_TRANSFER functions.

6.3.5 Hooking DEFINE_VECTOR_EXCHANGE_PROPERTY UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_VECTOR_EXCHANGE_RATE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the User-Defined Functions dialog box (Figure 6.3.9) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first select Mixture from the Model list of the Multiphase Model dialog box, and make sure that the Slip Velocity option is enabled.

⬧ Models ⟶ ☰ Multiphase ⟶ Edit...

Next, open the Phase Interaction dialog box (Figure 6.3.8).

✧ Phases ⟶ Interaction...
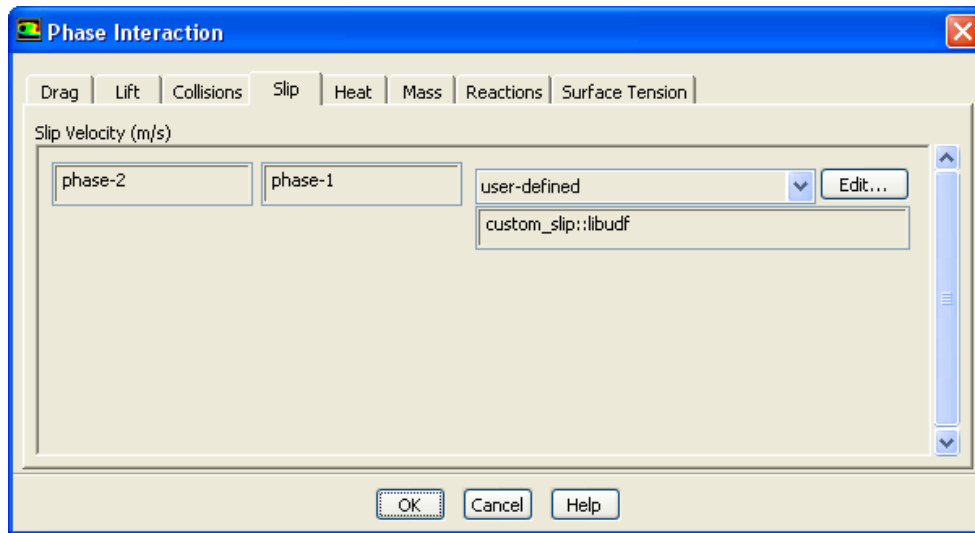
Figure 6.3.8: The Phase Interaction Dialog Box

Click the Slip tab in the Phase Interaction dialog box, and select user-defined from the drop-down list for the Slip Velocity. This will open the User-Defined Functions dialog box.
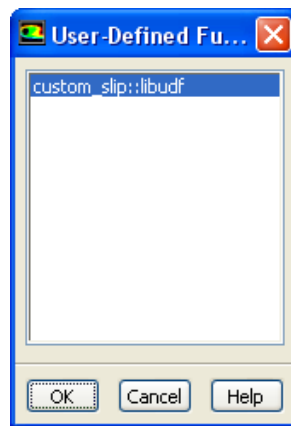
Figure 6.3.9: The User-Defined Functions Dialog Box

Finally, select the function name (e.g., custom_slip::libudf) from the list of UDFs displayed in the User-Defined Functions dialog box (Figure 6.3.9), and click OK. The UDF name will appear in the text box below the Slip Velocity drop-down list in the Phase Interaction dialog box. Click OK to close the Phase Interaction dialog box.

See Section 2.4.5: DEFINE_VECTOR_EXCHANGE_PROPERTY for details about DEFINE_VECTOR_EXCHANGE_PROPERTY functions.

6.4   Hooking Discrete Phase Model (DPM) UDFs
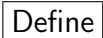
This section contains methods for hooking UDFs to ANSYS FLUENT that have been

- defined using `DEFINE` macros described in Section 2.5: Discrete Phase Model (DPM) `DEFINE` Macros, and

- interpreted or compiled using methods described in Chapters 4 or 5, respectively.

6.4.1   Hooking `DEFINE_DPM_BC` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_BC` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the appropriate boundary condition dialog box (Figure 6.4.1) in ANSYS FLUENT.

To hook the UDF, first create an injection using the Injections dialog box.

Define ⟶Injections...

Next, open the Boundary Conditions task page.

⬙ Boundary Conditions

Select the boundary in the Zone list and click Edit... to open the boundary condition dialog box (e.g., the Wall dialog box, as shown in Figure 6.4.1).
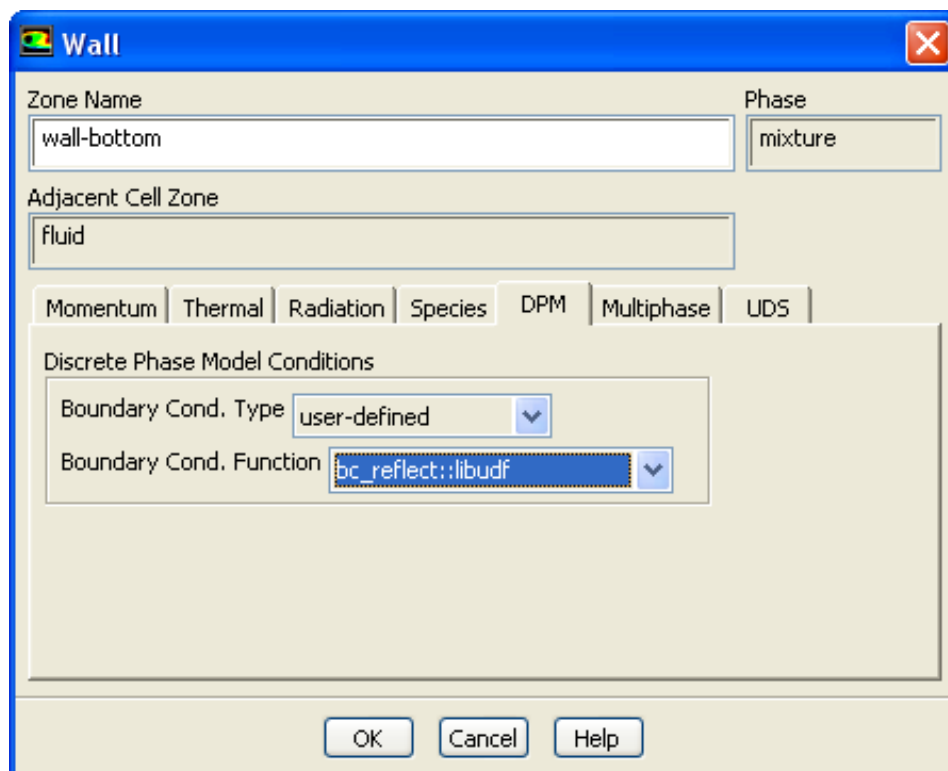
Figure 6.4.1: The Wall Dialog Box

Click the DPM tab and select user_defined from the Boundary Cond. Type drop-down list in the Discrete Phase Model Conditions group box. This will expand the dialog box to allow you to select the function name (e.g., bc_reflect::libudf) from the Boundary Cond. Function drop-down list (Figure 6.4.1). Click OK.

See Section 2.5.1: DEFINE_DPM_BC for details about DEFINE_DPM_BC functions.

6.4.2   Hooking DEFINE_DPM_BODY_FORCE **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DPM_BODY_FORCE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Discrete Phase Model dialog box (Figure 6.4.2) in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

⟡ Models ⟶ ≣ Discrete Phase ⟶ Edit...
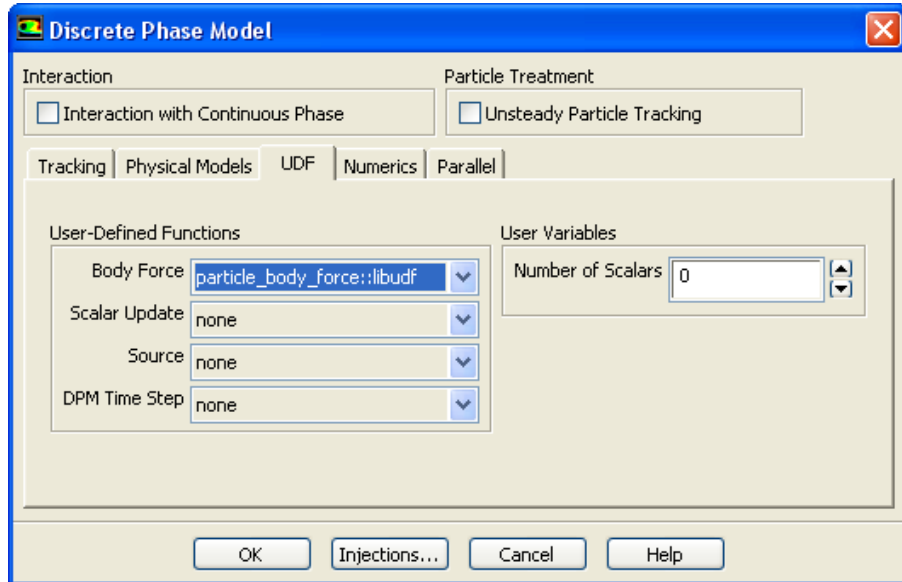
Figure 6.4.2: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, click the UDF tab in the Discrete Phase Model dialog box. Select the function name (e.g., particle_body_force::libudf) from the Body Force drop-down list under User-Defined Functions (Figure 6.4.2), and click OK.

See Section 2.5.2: DEFINE_DPM_BODY_FORCE for details about DEFINE_DPM_BODY_FORCE functions.

6.4.3 Hooking `DEFINE_DPM_DRAG` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_DRAG` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Discrete Phase Model dialog box (Figure 6.4.3) in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

⬦ Models ⟶ 𝌆 Discrete Phase ⟶ Edit...
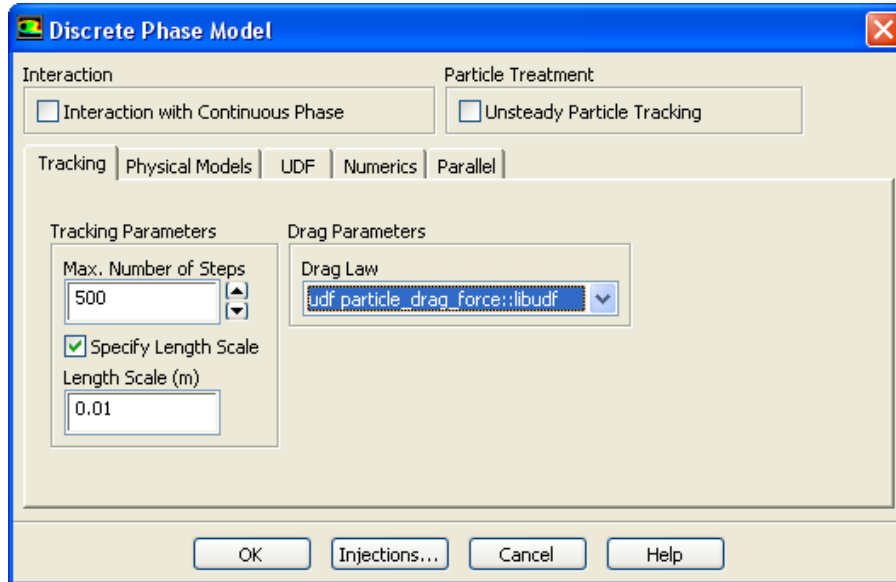
Figure 6.4.3: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, click the Tracking tab in the Discrete Phase Model dialog box. Select the function name (e.g., particle_drag_force::libudf) from the Drag Law drop-down list in the Drag Parameters group box (Figure 6.4.3), and click OK. (Note that function names listed in the drop-down list are preceded by the word udf, as in udf particle_drag_force::libudf.)

See Section 2.5.3: `DEFINE_DPM_DRAG` for details about `DEFINE_DPM_DRAG` functions.

6.4.4   Hooking DEFINE_DPM_EROSION UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DPM_EROSION UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Discrete Phase Model dialog box (Figure 6.4.4) in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

✥ Models ⟶ ☰ Discrete Phase ⟶ Edit...
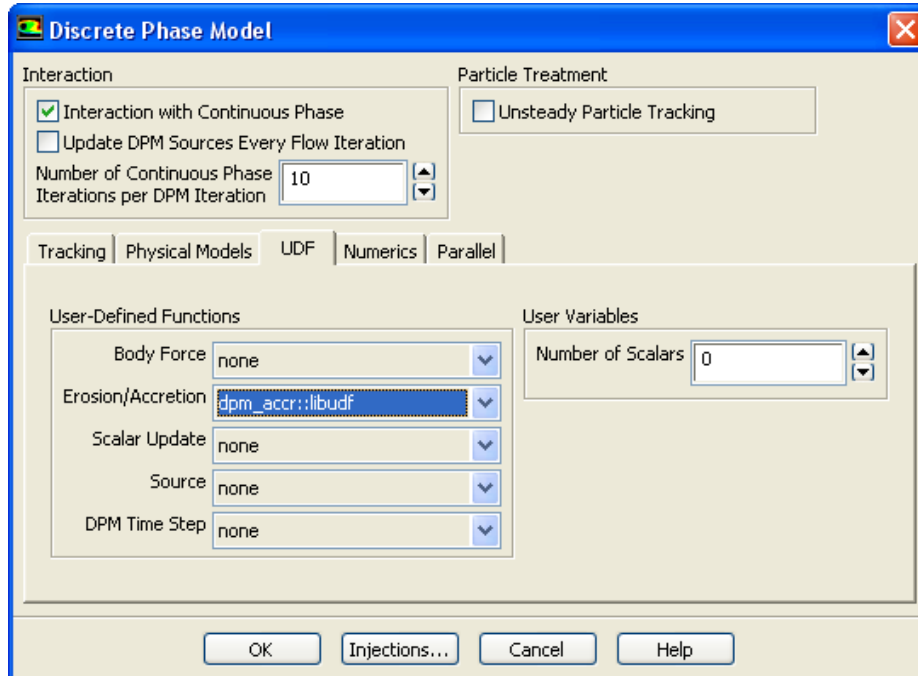
Figure 6.4.4: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, enable the Interaction with Continuous Phase option under Interaction in the Discrete Phase Model dialog box. Then, click the Physical Models tab and enable the Erosion/Accretion option. Finally, click the UDF tab and select the function name (e.g., dpm_accr::libudf) from the Erosion/Accretion drop-down list in the User-Defined Functions group box (Figure 6.4.3), and click OK.

See Section 2.5.4: DEFINE_DPM_EROSION for details about DEFINE_DPM_EROSION functions.

6.4.5   Hooking `DEFINE_DPM_HEAT_MASS` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_HEAT_MASS` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Set Injection Properties dialog box (Figure 6.4.5) in ANSYS FLUENT.

To hook the UDF, first set up your species transport model in the Species Model dialog box.

⟡ Models ⟶ ☰ Species ⟶ Edit...

Select Species Transport from the Model list and click OK.

Next, create a particle injection in the Injections dialog box.

Define ⟶Injections...

Click Create in the Injections dialog box to open the Set Injection Properties dialog box (Figure 6.4.5).
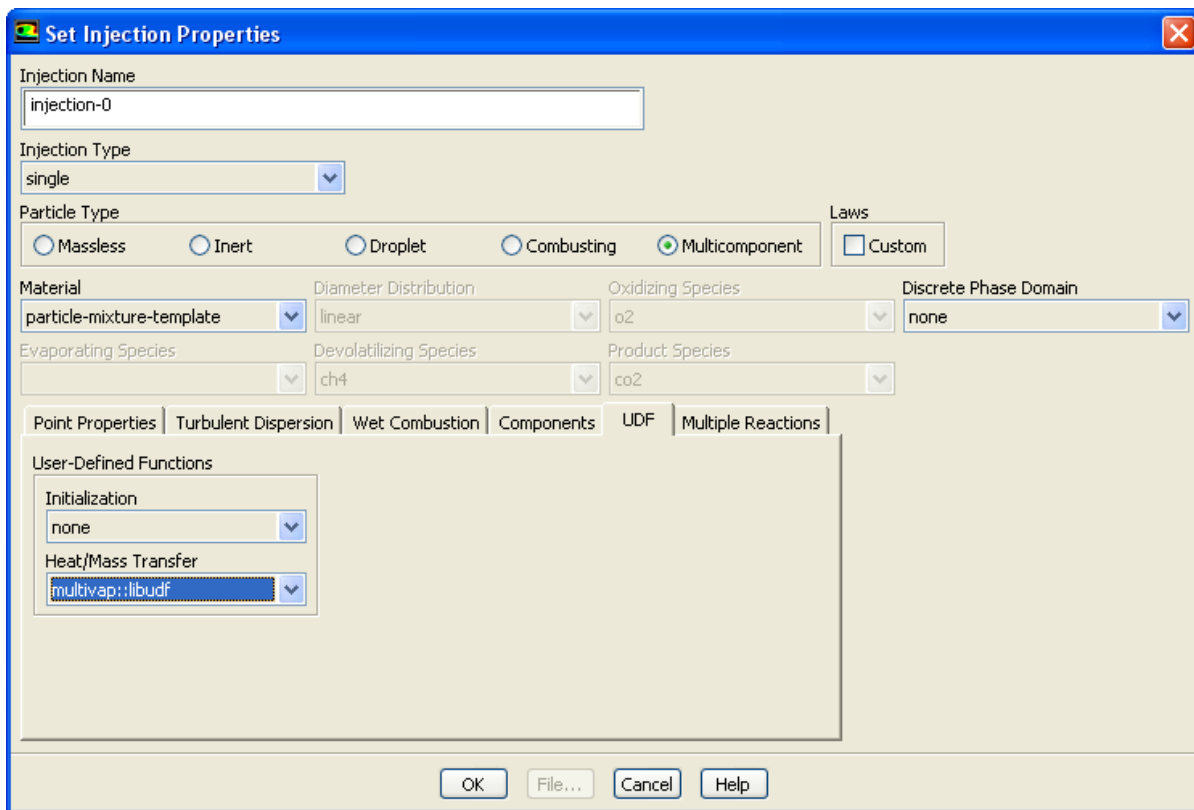
Figure 6.4.5: The Set Injections Dialog Box
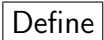
Set up the particle injection in the Set Injection Properties dialog box, being sure to select Multicomponent in the Particle Type group box. Then click the UDF tab, and select the function name (e.g., multivap::libudf) from the Heat/Mass Transfer drop-down list in the User-Defined Functions group box. Click OK.

See Section 2.5.6: DEFINE_DPM_INJECTION_INIT for details about DEFINE_DPM_INJECTION_INIT functions.

6.4.6 Hooking DEFINE_DPM_INJECTION_INIT **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DPM_INJECTION_INIT UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Set Injection Properties dialog box (Figure 6.4.6) in ANSYS FLUENT.

To hook the UDF, first create a particle injection in the Injections dialog box.

$\boxed{\text{Define}}$ ⟶Injections...

Click Create in the Injections dialog box to open the Set Injection Properties dialog box (Figure 6.4.6.

Set up the particle injection in the Set Injection Properties dialog box. Then click the UDF tab and select the function name (e.g., init_bubbles::libudf) from the Initialization drop-down list under User-Defined Functions. Click OK.

See Section 2.5.6: DEFINE_DPM_INJECTION_INIT for details about DEFINE_DPM_INJECTION_INIT functions.
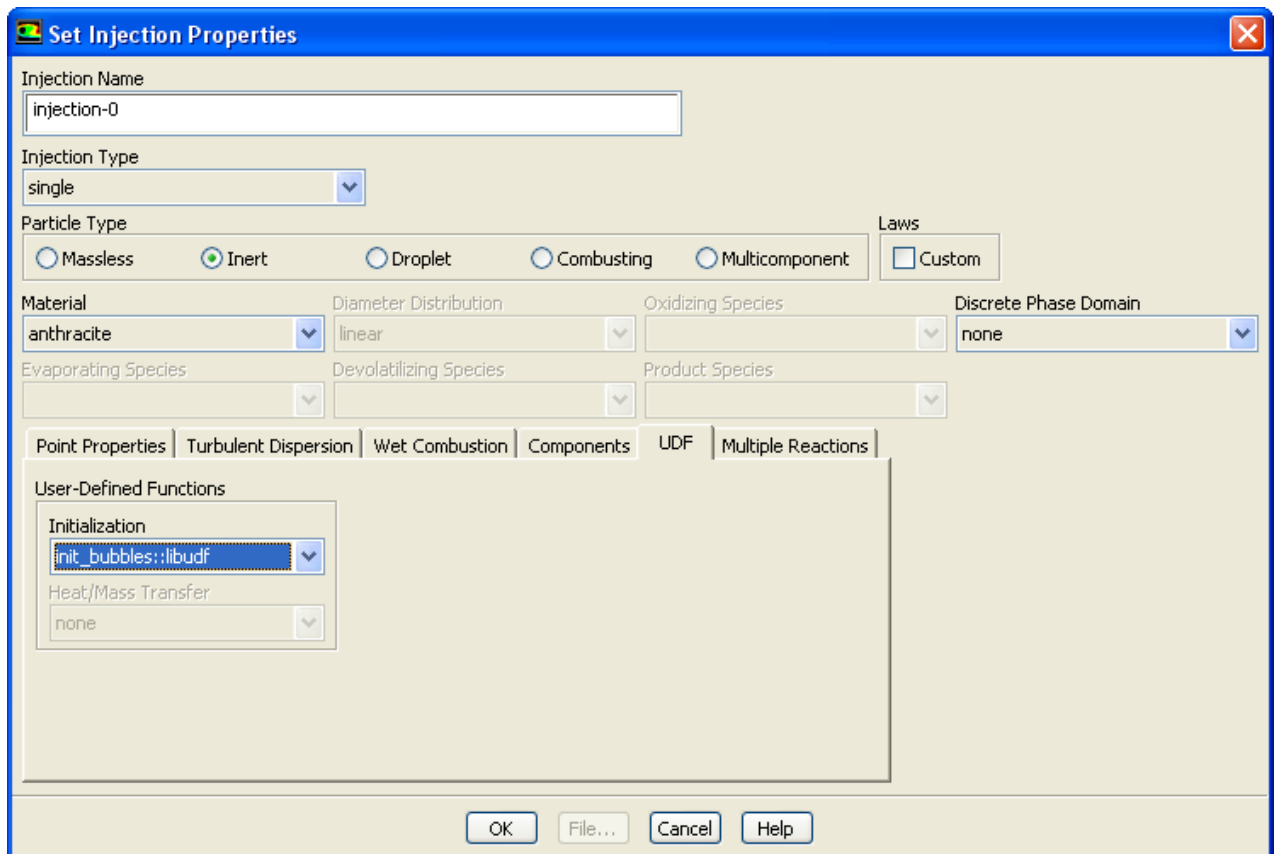
Figure 6.4.6: The Injections Dialog Box

6.4.7 Hooking `DEFINE_DPM_LAW` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_LAW` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Custom Laws dialog box (Figure 6.4.7) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first create a particle injection in the Injections dialog box.

Define ⟶Injections...

Click Create in the Injections dialog box to open the Set Injection Properties dialog box. Then, enable the Custom option in the Laws group box in the Set Injection Properties dialog box, in order to open the Custom Laws dialog box (Figure 6.4.7).
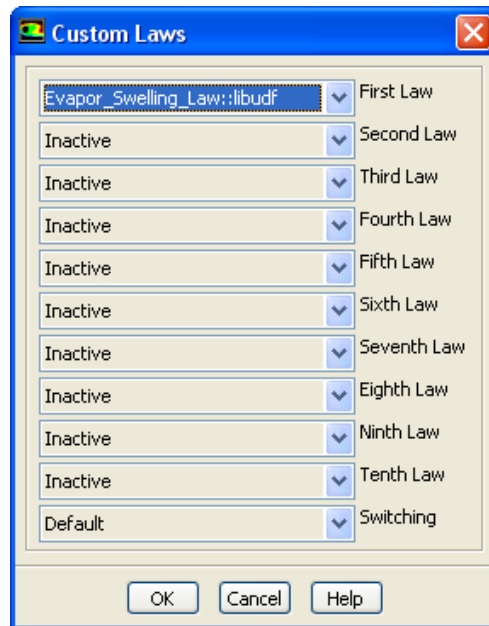
Figure 6.4.7: The Custom Laws Dialog Box

In the Custom Laws dialog box, select the function name (e.g., Evapor_Swelling_Law::libudf) from the appropriate drop-down list located to the left of each of the six particle laws (e.g., First Law), and click OK.

See Section 2.5.7: `DEFINE_DPM_LAW` for details about `DEFINE_DPM_LAW` functions.

6.4.8 **Hooking** DEFINE_DPM_OUTPUT **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DPM_OUTPUT UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Sample Trajectories dialog box (Figure 6.4.8) in ANSYS FLUENT.

In order to hook the UDF, you must first set up the discrete phase model (e.g., create a particle injection). After you have run the calculation and generated data, open the Sample Trajectories dialog box (Figure 6.4.8).

⬥ Reports ⟶ ☰ Sample ⟶ Set Up...
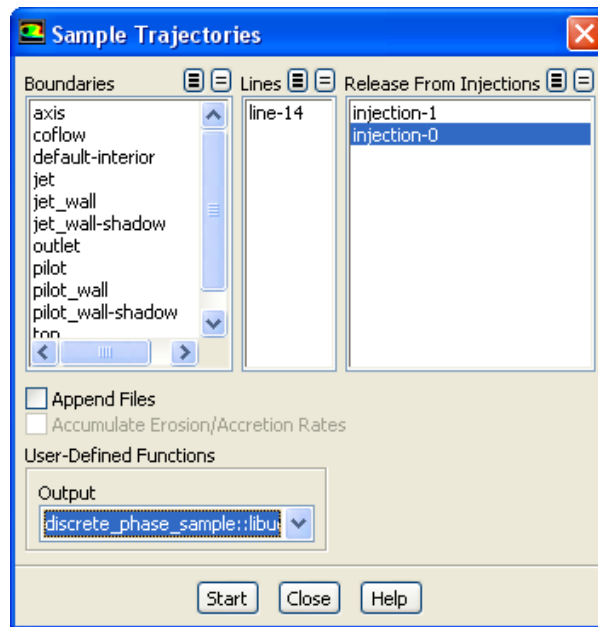
Figure 6.4.8: The Sample Trajectories Dialog Box

Select the appropriate injection in the Release From Injections list, indicate where the sample will be written (e.g., make a selection in the Lines list), and select the function name (e.g., discrete_phase_sample::libudf) from the Output drop-down list under User-Defined Functions. Then click Compute (for steady calculations) or Start (for transient calculations).

See Section 2.5.8: DEFINE_DPM_OUTPUT for details about DEFINE_DPM_OUTPUT functions.

6.4.9 Hooking `DEFINE_DPM_PROPERTY` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_PROPERTY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the User-Defined Functions dialog box.

To hook the UDF to ANSYS FLUENT, you will first need to open the Materials task page.

⬦ Materials

Select a material from Materials list and click the Create/Edit... button to open the Create/Edit Materials dialog box (Figure 6.4.9).

Figure 6.4.9: The Create/Edit Materials Dialog Box

Select user-defined in the drop-down list for one of the properties (e.g., Particle Emissivity) in the Create/Edit Materials dialog box, in order to open the User-Defined Functions dialog box (Figure 6.4.10).

i In order for the Particle Emissivity property to be displayed in the sample dialog box shown above, you must enable a radiation model, turn on the Particle Radiation Interaction option in the Discrete Phase Model dialog box, and introduce a particle injection in the Injections dialog box.

Figure 6.4.10: The User-Defined Functions Dialog Box

Select the function name (e.g., coal_emissivity::libudf) from the list of UDFs displayed in the User-Defined Functions dialog box, and click OK. The name of the function will subsequently be displayed under the selected property (e.g., Particle Emissivity) in the Create/Edit Materials dialog box.

See Section 2.3.16: DEFINE_PROPERTY UDFs for details about DEFINE_DPM_PROPERTY functions.

6.4.10 Hooking DEFINE_DPM_SCALAR_UPDATE UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DPM_SCALAR_UPDATE UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Discrete Phase Model dialog box (Figure 6.4.11) in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

⊕ Models ⟶ ☰ Discrete Phase ⟶ Edit...
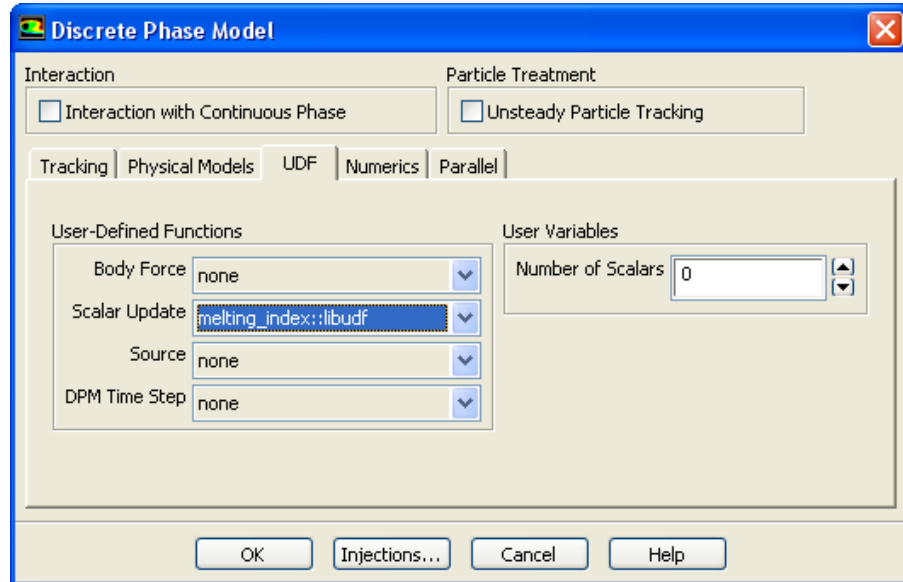
Figure 6.4.11: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, click the UDF tab in the Discrete Phase Model dialog box. Select the function name (e.g., melting_index::libudf) from the Scalar Update drop-down list under User-Defined Functions (Figure 6.4.11), and click OK.

See Section 2.5.10: DEFINE_DPM_SCALAR_UPDATE for details about DEFINE_DPM_SCALAR_UPDATE functions.

6.4.11   **Hooking** `DEFINE_DPM_SOURCE` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Discrete Phase Model dialog box (Figure 6.4.12) in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

⬥ Models ⟶ ☰ Discrete Phase ⟶ Edit...
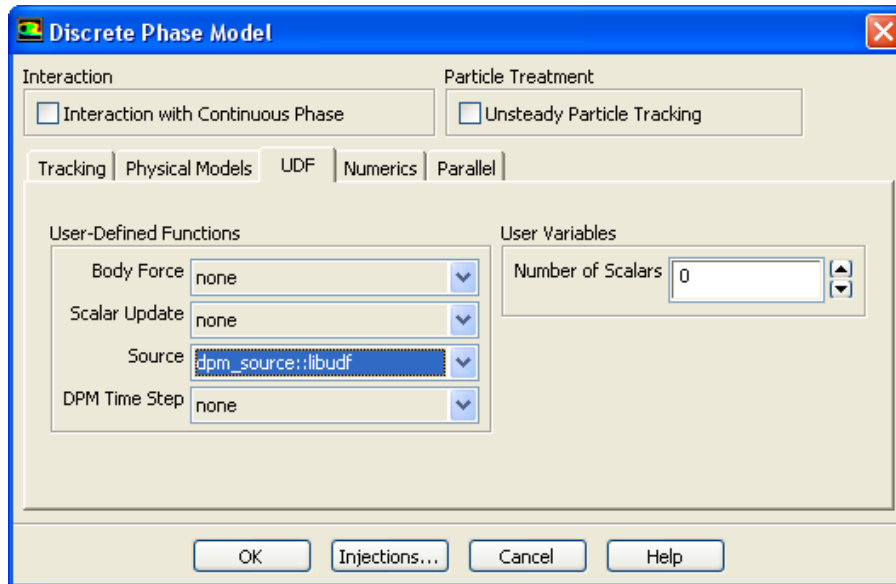
Figure 6.4.12: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, click the UDF tab in the Discrete Phase Model dialog box. Select the function name (e.g., dpm_source::libudf) from the Source drop-down list under User-Defined Functions (Figure 6.4.12), and click OK.

See Section 2.5.11: `DEFINE_DPM_SOURCE` for details about `DEFINE_DPM_SOURCE` functions.

6.4.12 Hooking `DEFINE_DPM_SPRAY_COLLIDE` UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_SPRAY_COLLIDE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Discrete Phase Model dialog box (Figure 6.4.13) in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

⬖ Models ⟶ ≣ Discrete Phase ⟶ Edit...
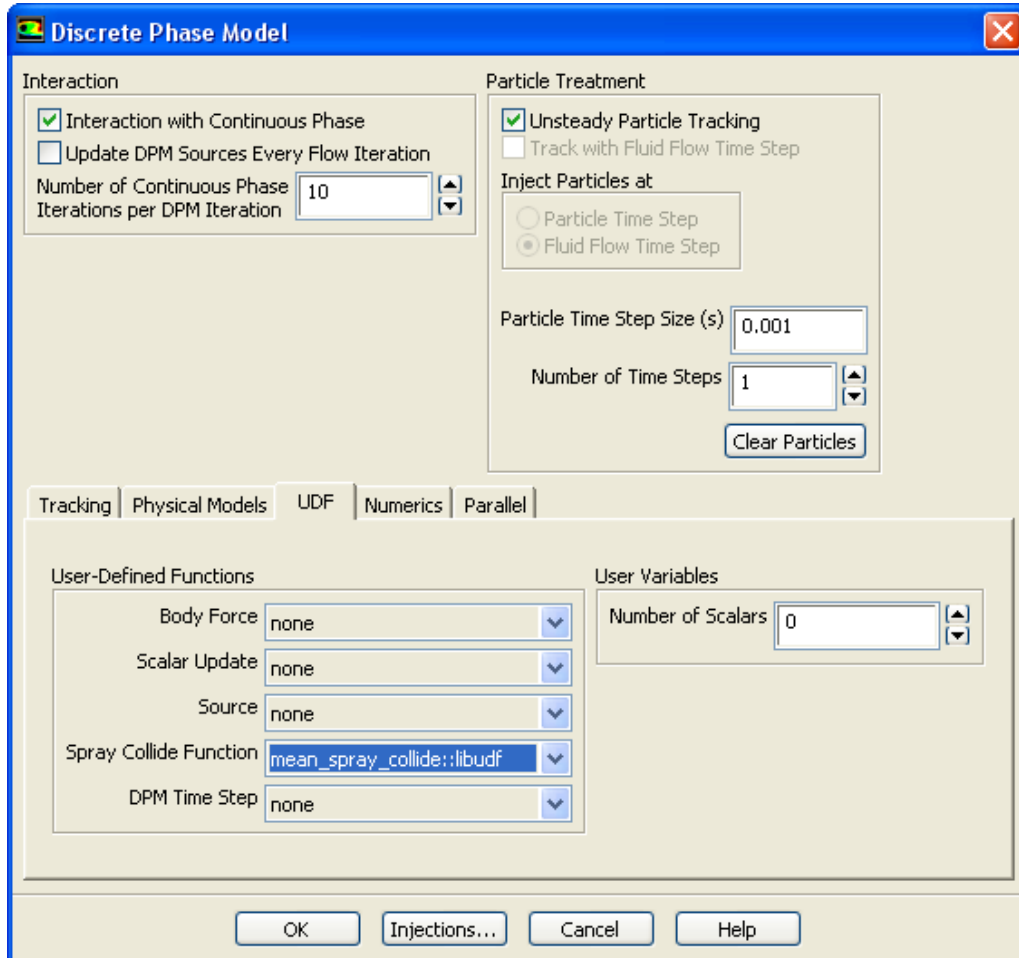
Figure 6.4.13: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, click the Physical Models tab in the Discrete Phase Model dialog box and enable the Droplet Collision option in the Spray Model group box. Then, click the UDF tab and

select the function name (e.g., mean_spray_collide::libudf) from the Spray Collide Function drop-down list in the User-Defined Functions group box (Figure 6.4.3), and click OK.

See Section 2.5.12: `DEFINE_DPM_SPRAY_COLLIDE` for details about `DEFINE_DPM_SPRAY_COLLIDE` functions.

6.4.13   **Hooking** `DEFINE_DPM_SWITCH` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_SWITCH` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Custom Laws dialog box (Figure 6.4.14) in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, first create a particle injection in the Injections dialog box.

Define ⟶Injections...

Click Create in the Injections dialog box to open the Set Injection Properties dialog box, so that you can set up the injection properties. Enable the Custom option in the Laws group box in the Set Injection Properties dialog box, in order to open the Custom Laws dialog box (Figure 6.4.14).
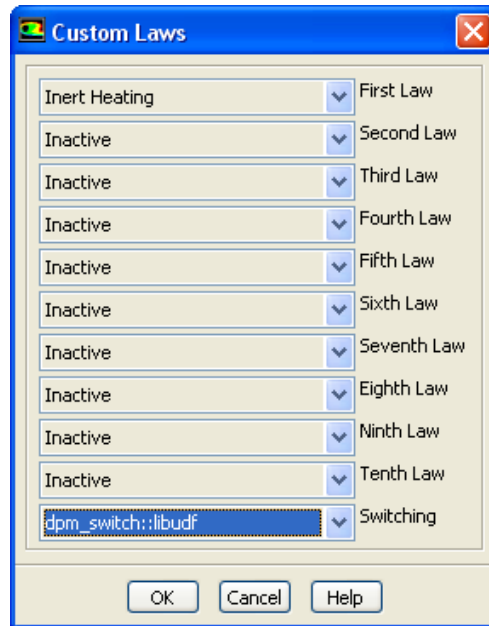
Figure 6.4.14: The Custom Laws Dialog Box

In the Custom Laws dialog box, select the function name (e.g., dpm_switch::libudf) from the Switching drop-down list and click OK.

See Section 2.5.13: DEFINE_DPM_SWITCH for details about DEFINE_DPM_SWITCH functions.

6.4.14 Hooking DEFINE_DPM_TIMESTEP **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_DPM_TIMESTEP UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in ANSYS FLUENT.

To hook the UDF, first open the Discrete Phase Model dialog box.

✧ Models ⟶ ≣ Discrete Phase ⟶ Edit...
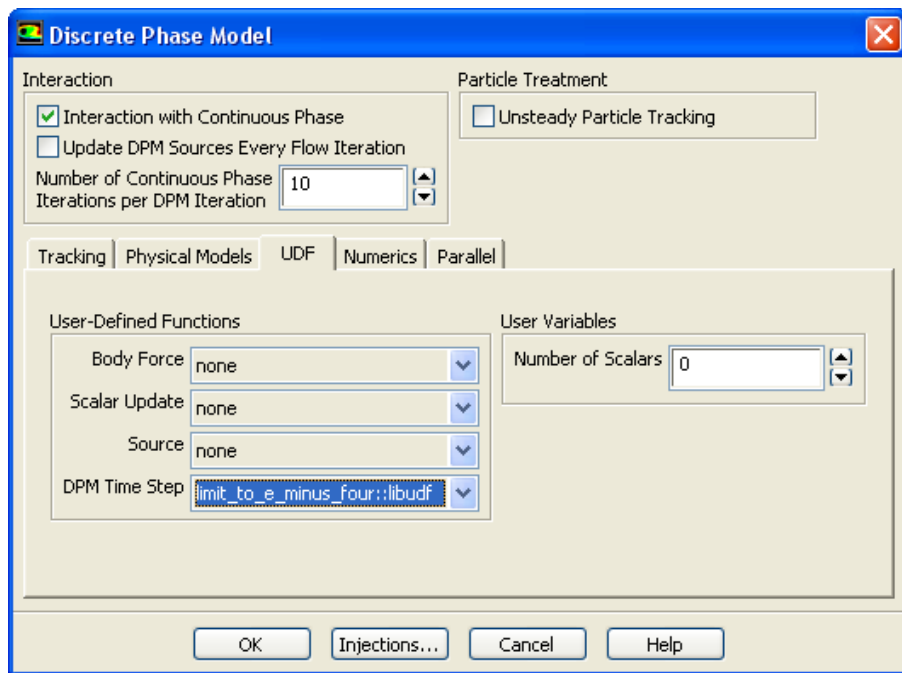
Figure 6.4.15: The Discrete Phase Model Dialog Box

Click the Injections... button to open the Injections dialog box. Create an injection and then click Close in the Injections dialog box.

Next, click the UDF tab in the Discrete Phase Model dialog box. Select the function name (e.g., limit_to_e_minus_four::libudf) from the DPM Time Step drop-down list under User-Defined Functions (Figure 6.4.15), and click OK.

See Section 2.5.14: DEFINE_DPM_TIMESTEP for details about DEFINE_DPM_TIMESTEP functions.

6.4.15   Hooking `DEFINE_DPM_VP_EQUILIB` **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your `DEFINE_DPM_VP_EQUILIB` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable from the **Create/Edit Materials** dialog box in ANSYS FLUENT.

To hook the UDF, first set up your species transport and combustion models in the **Species Model** dialog box.

⟐ Models ⟶ ≣ Species ⟶ Edit...

Then, create a particle injection using the **Injections** dialog box.

Define ⟶Injections...

Click **Create** in the **Injections** dialog box to open the **Set Injection Properties** dialog box. Set up the particle injection, making sure to select **Multicomponent** for the **Particle Type**.

Next, open the **Materials** task page.

⟐ Materials

Select the appropriate material in the **Materials** list (e.g., **particle-mixture-template**) and click **Create/Edit** to open the **Create/Edit Materials** dialog box (Figure 6.4.16).
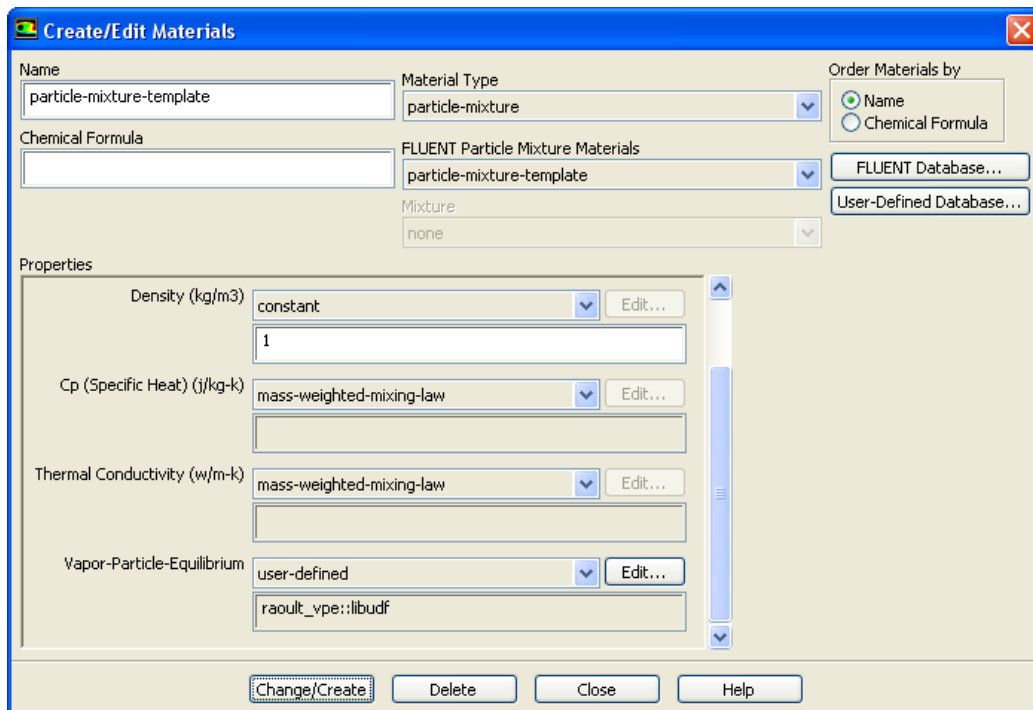
Figure 6.4.16: The **Create/Edit Materials** Dialog Box

Select user-defined from the drop-down list for Vapor-Particle-Equilibrium in the Properties group box. This will open the User-Defined Functions dialog box. Select the UDF name (e.g., raoult_vp::libudf) from the list of UDFs displayed and click OK. Then click Change/Create in the Create/Edit Materials dialog box.
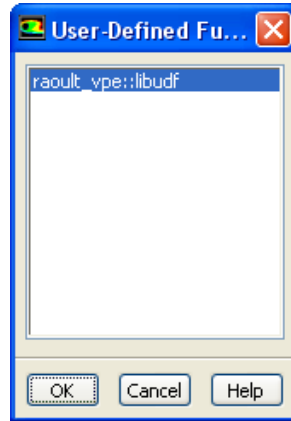
Figure 6.4.17: The User-Defined Functions Dialog Box

See Section 2.5.15: `DEFINE_DPM_VP_EQUILIB` for details about `DEFINE_DPM_VP_EQUILIBRIUM` functions.

6.5 Hooking Dynamic Mesh UDFs

This section contains methods for hooking UDFs to ANSYS FLUENT that have been defined using `DEFINE` macros described in Section 2.6: Dynamic Mesh `DEFINE` Macros, and interpreted or compiled using methods described in Chapters 4 or 5, respectively.

6.5.1 Hooking `DEFINE_CG_MOTION` UDFs

After you have compiled (Chapter 5: Compiling UDFs) your `DEFINE_CG_MOTION` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the Dynamic Mesh Zones dialog box (Figure 6.5.1).

To hook the UDF to ANSYS FLUENT, you will first need to enable the Dynamic Mesh option in the Dynamic Mesh task page.

✥ Dynamic Mesh ⟶ ☑ Dynamic Mesh

Next, open the Dynamic Mesh Zones dialog box.
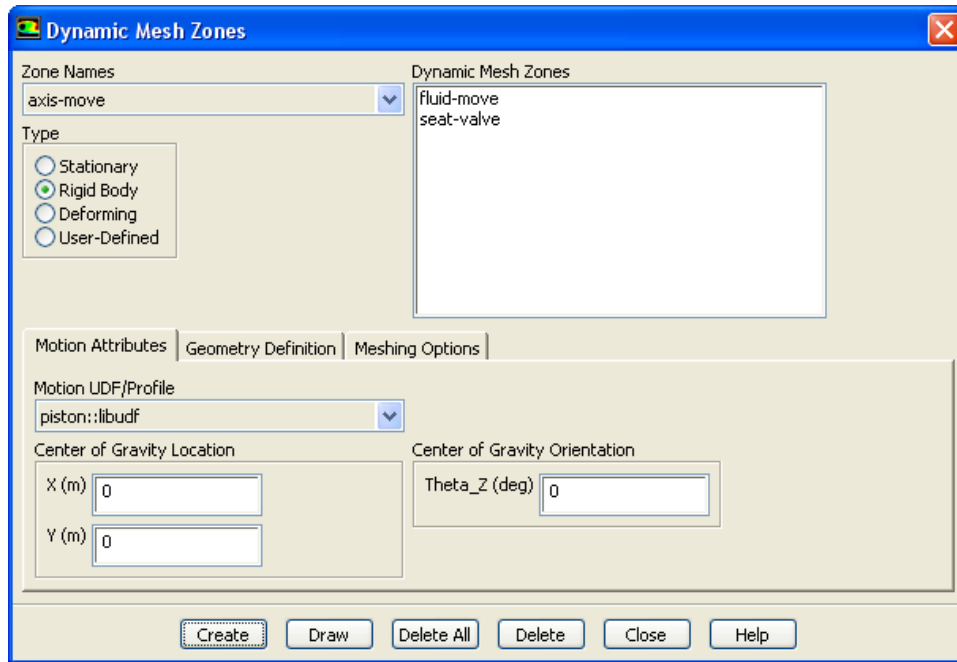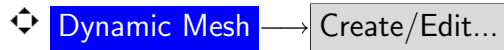
Figure 6.5.1: The Dynamic Mesh Zones Dialog Box

Select Rigid Body under Type in the Dynamic Mesh Zones dialog box (Figure 6.5.1) and click the Motion Attributes tab. Finally, select the function name (e.g., piston::libudf) from the Motion UDF/Profile drop-down list, and click Create then Close.

See Section 2.6.1: DEFINE_CG_MOTION for details about DEFINE_CG_MOTION functions.

6.5.2 **Hooking** DEFINE_DYNAMIC_ZONE_PROPERTY **UDFs**

The DEFINE_DYNAMIC_ZONE_PROPERTY UDF can be hooked in order to define the following:

- the swirl center for in-cylinder applications

- a variable cell layering height

Hooking a Swirl Center UDF

After you have complied your DEFINE_DYNAMIC_ZONE_PROPERTY UDF (as described in Chapter 5: Compiling UDFs), the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the In-Cylinder Output Controls dialog box (Figure 6.5.2).

To hook the UDF to ANSYS FLUENT, you will first need to select Transient from the Time list in the Solver group box of the General task page.

✧ General ⟶ ⦿ Transient

Next, enable the Dynamic Mesh option in the Dynamic Mesh task page.
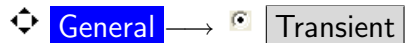
✧ Dynamic Mesh ⟶ ☑ Dynamic Mesh
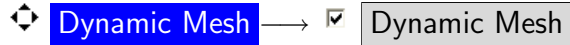
Then, enable the In-Cylinder option in the Options group box, and click the Settings button to open the In-Cylinder Settings dialog box. After you have updated the parameters in this dialog box, click the Output Controls... button to open the In-Cylinder Output Controls dialog box (Figure 6.5.2).
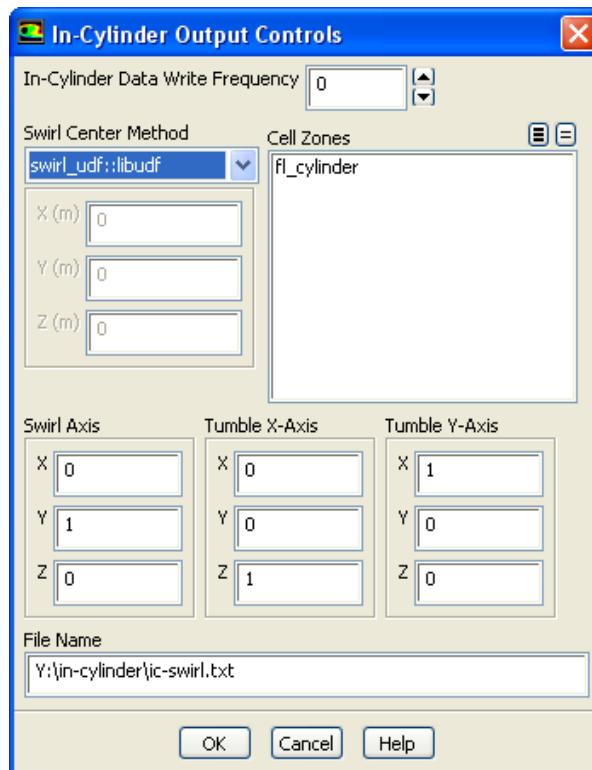
Figure 6.5.2: In-Cylinder Output Controls Dialog Box

Select the UDF library (e.g., swirl_udf::libudf) from the Swirl Center Method drop-down list in the In-Cylinder Output Controls dialog box. Click OK and close the In-Cylinder Output Controls dialog box.

See Section 2.6.2: DEFINE_DYNAMIC_ZONE_PROPERTY for further details about DEFINE_DYNAMIC_ZONE_PROPERTY functions.

Hooking a Variable Cell Layering Height UDF

After you have compiled your DEFINE_DYNAMIC_ZONE_PROPERTY UDF (as described in Chapter 5: Compiling UDFs), the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Dynamic Mesh Zones dialog box (Figure 6.5.3).

> i Since the DEFINE_DYNAMIC_ZONE_PROPERTY UDF is a function of time or crank angle, you must make sure that you have selected Transient from the Time list in the Solver group box of the General task page before proceeding.

To hook the UDF to ANSYS FLUENT, you will first need to enable the Dynamic Mesh option in the Dynamic Mesh task page.

⬥ Dynamic Mesh ⟶ ☑ Dynamic Mesh

Then, enable the Layering option in the Mesh Methods list, and click the Settings... button to open the Mesh Methods Settings dialog box. In the Layering tab, select Height Based from the Options list, and set the Split Factor and Collapse Factor to appropriate values. Then click OK.

Next, specify the meshing options in the Dynamic Mesh Zones dialog box (Figure 6.5.3).

⬥ Dynamic Mesh ⟶ Create/Edit...
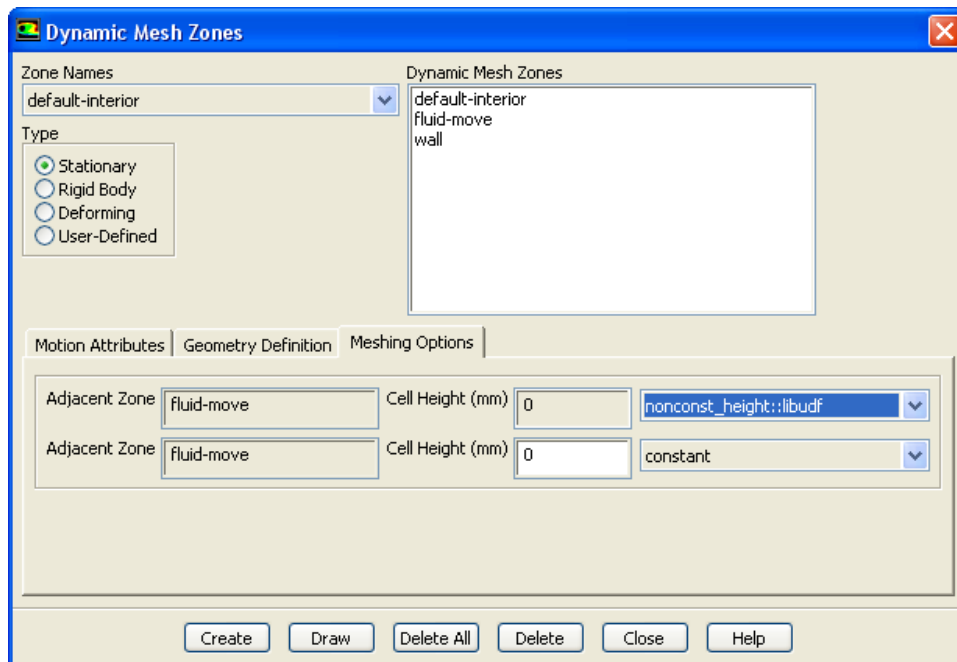
Figure 6.5.3: The Dynamic Mesh Zones Dialog Box

Select Stationary, Rigid Body, or User-Defined from the Type list in the Dynamic Mesh Zones dialog box. Click the Meshing Options tab, and select the UDF library (e.g., nonconst_height::libudf) from the Cell Height drop-down list. Finally, click Create and close the Dynamic Mesh Zones dialog box.

See Section 2.6.2: DEFINE_DYNAMIC_ZONE_PROPERTY for further details about DEFINE_DYNAMIC_ZONE_PROPERTY functions.

6.5.3   Hooking DEFINE_GEOM UDFs

After you have compiled (Chapter 5: Compiling UDFs) your DEFINE_GEOM UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Dynamic Mesh Zones dialog box.

To hook the UDF to ANSYS FLUENT, you will first need to enable the Dynamic Mesh option in the Dynamic Mesh task page.

✛ Dynamic Mesh ⟶ ☑ Dynamic Mesh

Next, open the Dynamic Mesh Zones dialog box (Figure 6.5.4).

✛ Dynamic Mesh ⟶ Create/Edit...
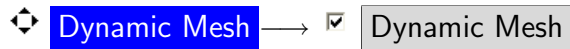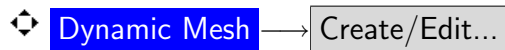
Select Deforming under Type in the Dynamic Mesh Zones dialog box (Figure 6.5.4) and click the Geometry Definition tab. Select user-defined in the Definition drop-down list, and select the function name (e.g., parabola::libudf) from the Geometry UDF drop-down list. Click Create and then Close.

See Section 2.6.3: DEFINE_GEOM for details about DEFINE_GEOM functions.
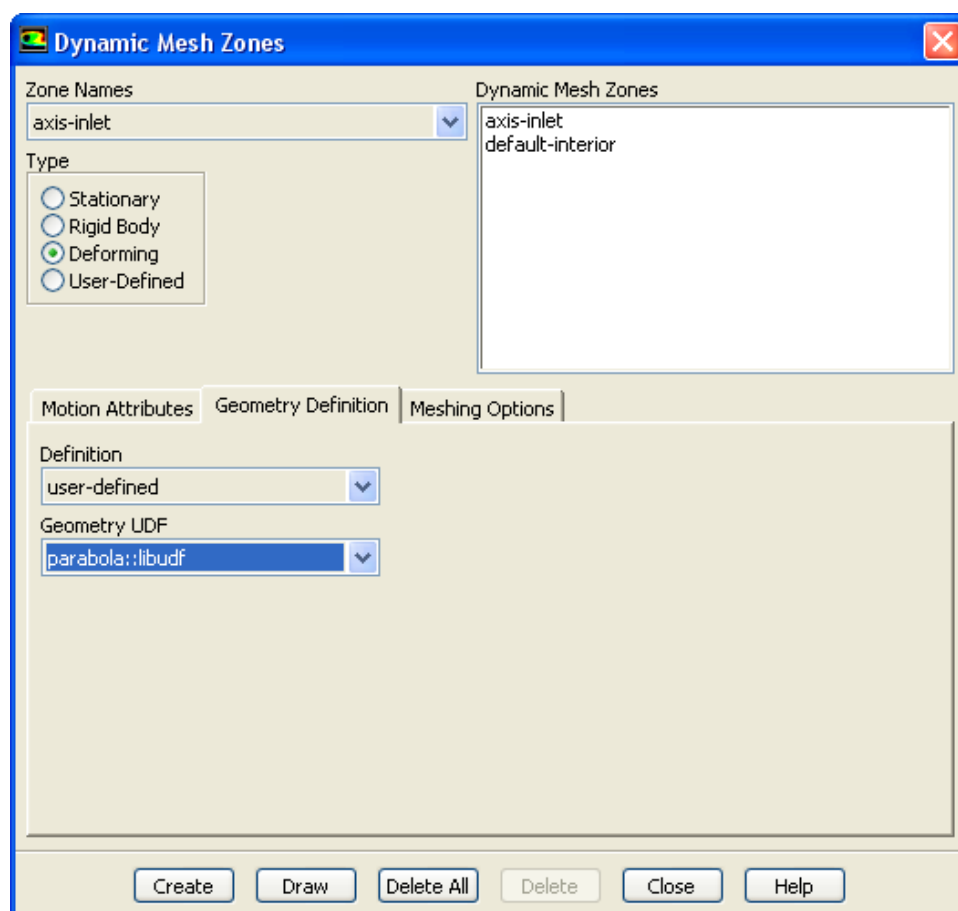
Figure 6.5.4: The **Dynamic Mesh Zones** Dialog Box

6.5.4 Hooking DEFINE_GRID_MOTION UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_GRID_MOTION UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Dynamic Mesh Zones dialog box (Figure 6.5.5).

To hook the UDF to ANSYS FLUENT, you will first need to enable the Dynamic Mesh option in the Dynamic Mesh task page.

✧ Dynamic Mesh $\longrightarrow$ ☑ Dynamic Mesh

Next, open the Dynamic Mesh Zones dialog box.

✧ Dynamic Mesh $\longrightarrow$ Create/Edit...
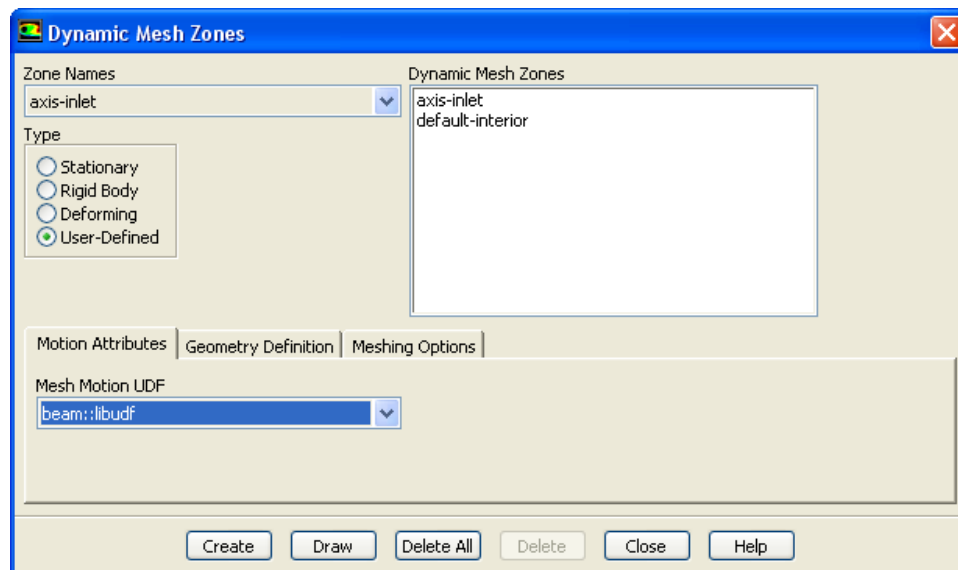
Figure 6.5.5: Dynamic Mesh Zones

Select User-Defined under Type in the Dynamic Mesh Zones dialog box (Figure 6.5.5) and click the Motion Attributes tab. Select the function name (e.g., beam::libudf) from the Mesh Motion UDF drop-down list. Click Create then Close.

See Section 2.6.4: DEFINE_GRID_MOTION for details about DEFINE_GRID_MOTION functions.

6.5.5   **Hooking** DEFINE_SDOF_PROPERTIES **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_SDOF_PROPERTIES UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable in the Dynamic Mesh Zones dialog box in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, you will first need to select Transient from the Time list in the Solver group box of the General task page.

✧ General ⟶ ⦿ Transient

Next, enable the Dynamic Mesh option in the Dynamic Mesh task page.
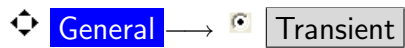
✧ Dynamic Mesh ⟶ ☑ Dynamic Mesh
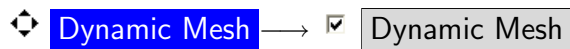
Then, enable the Six DOF option in the Options group box, and open the Dynamic Mesh Zones dialog box (Figure 6.5.6).

✧ Dynamic Mesh ⟶ Create/Edit...
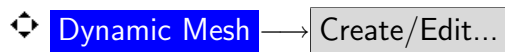
Select Rigid Body under Type in the Dynamic Mesh Zones dialog box (Figure 6.5.6) and click the Motion Attributes tab. Make sure that the On option in the Six DOF Solver Options group box is enabled, and select the function name (e.g., stage::libudf) from the Six DOF UDF drop-down list. Click Create then Close.

See Section 2.6.5: DEFINE_SDOF_PROPERTIES for details about DEFINE_SDOF_PROPERTIES functions.
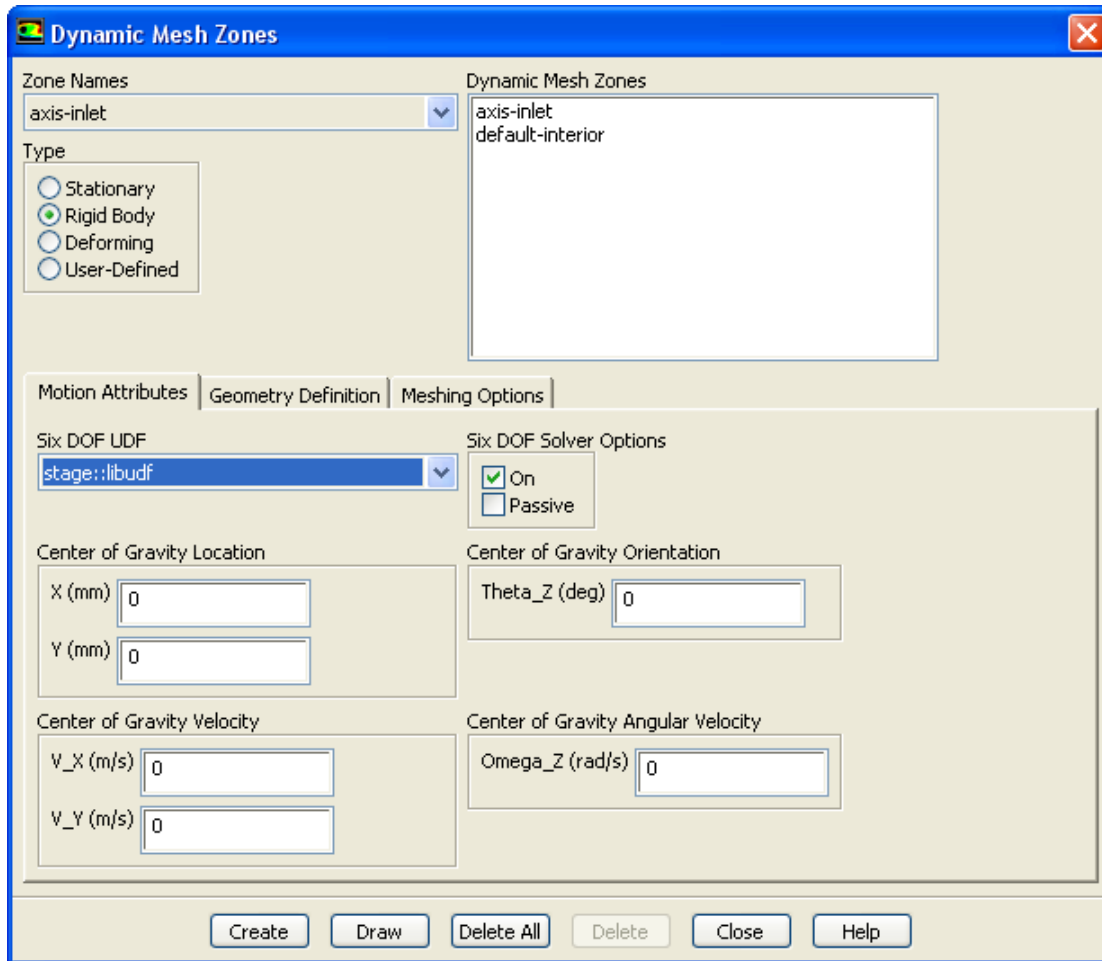
Figure 6.5.6: The Dynamic Mesh Zones Dialog Box

6.6 Hooking User-Defined Scalar (UDS) Transport Equation UDFs

This section contains methods for hooking anisotropic diffusion coefficient, flux, and unsteady UDFs for scalar equations that have been defined using DEFINE macros described in Section 2.7: User-Defined Scalar (UDS) Transport Equation DEFINE Macros and interpreted or compiled using methods described in Chapters 4 or 5, respectively. See Section 6.2.15: Hooking DEFINE_PROFILE UDFs, Section 6.2.19: Hooking DEFINE_SOURCE UDFs, and Section 6.2.3: Hooking DEFINE_DIFFUSIVITY UDFs to hook scalar source term, profile, or isotropic diffusion coefficient UDFs.

6.6.1 Hooking DEFINE_ANISOTROPIC_DIFFUSIVITY UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_ANISOTROPIC_DIFFUSIVITY UDF, the name of the function you supplied as the first DEFINE macro argument will become visible and selectable in ANSYS FLUENT.

To hook the UDF to ANSYS FLUENT, you will first need to open the User-Defined Scalars dialog box.

Define ⟶ User-Defined ⟶Scalars...

In the User-Defined Scalars dialog box, specify the Number of User-Defined Scalars (e.g., 2) and click OK. Next, open the Materials task page.

⬦ Materials

Select one of the materials in the Materials list and click Create/Edit... to open the Create/Edit Materials dialog box (Figure 6.6.1).
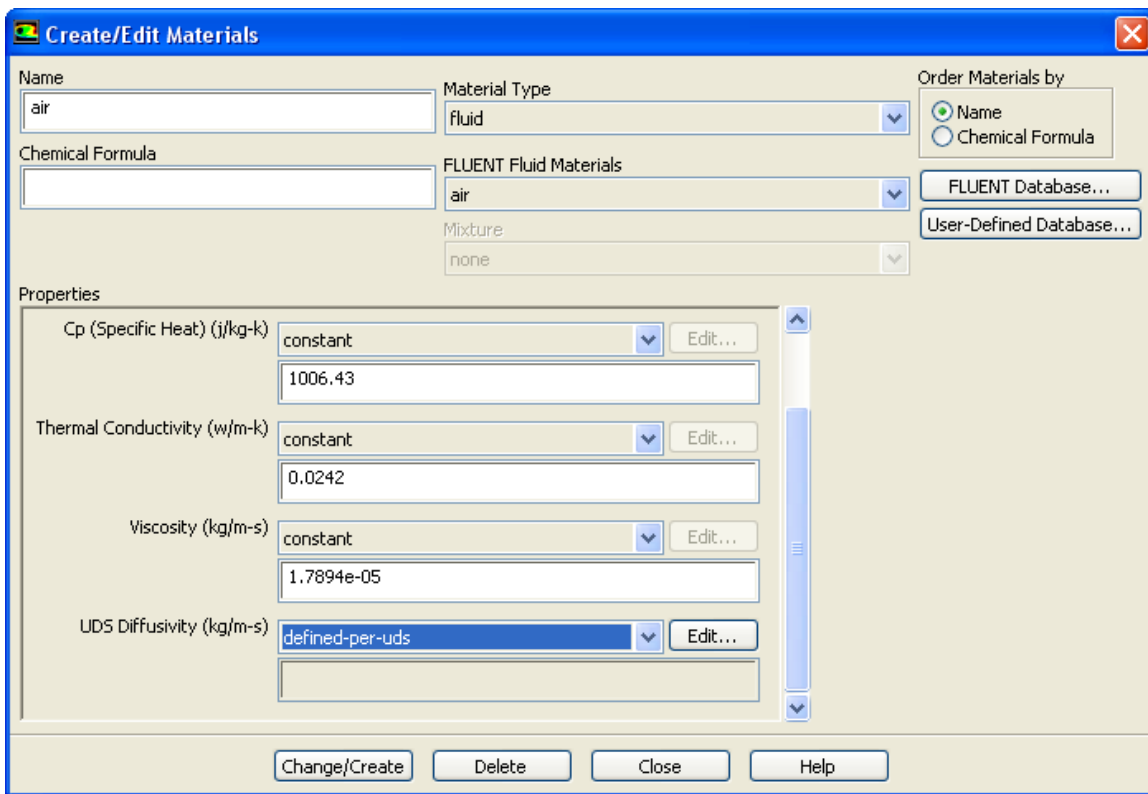
Figure 6.6.1: The Create/Edit Materials Dialog Box

Scroll down the Properties group box in the Create/Edit Materials dialog box (Figure 6.6.1), and select defined-per-uds from the UDS Diffusivity drop-down list. This will open the UDS Diffusion Coefficients dialog box (Figure 6.6.2).
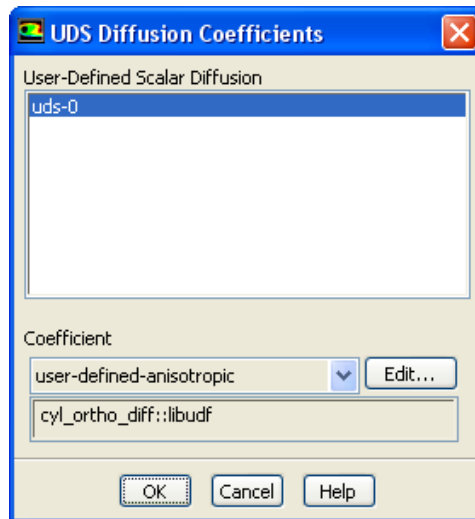
Figure 6.6.2: The UDS Diffusion Coefficients Dialog Box

In the UDS Diffusion Coefficients dialog box, select a scalar equation (e.g., uds-0) and select user-defined-anisotropic from the drop-down list under Coefficient. This will open the User-Defined Functions dialog box. Select the name of the UDF (e.g., cyl_ortho_diff::libudf) you wish to hook, and click OK. The name of the UDF will be displayed in the text box below the Coefficient drop-down list in the UDS Diffusion Coefficients dialog box. Click OK, and then click Change/Create in the Create/Edit Materials dialog box.

Note that you can hook a unique diffusion coefficient UDF for each scalar transport equation you have defined in your model.

See Section 2.7.2: DEFINE_ANISOTROPIC_DIFFUSIVITY for details about defining DEFINE_ANISOTROPIC_DIFFUSIVITY UDFs and the User's Guide for general information about UDS anisotropic diffusivity.

6.6.2   Hooking DEFINE_UDS_FLUX UDFs

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_UDS_FLUX UDF, the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the User-Defined Scalars dialog box (Figure 6.6.3) in ANSYS FLUENT.

Define ⟶ User-Defined ⟶Scalars...
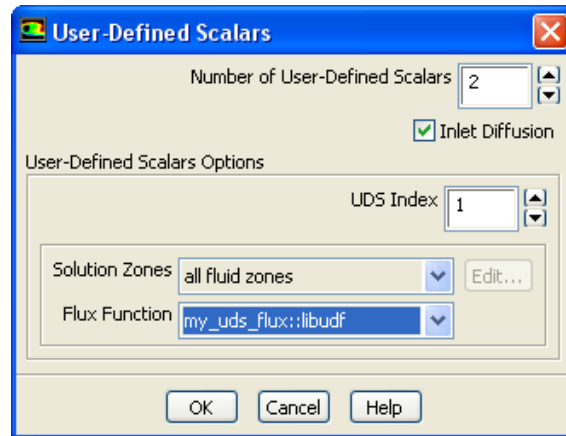
Figure 6.6.3: The User-Defined Scalars Dialog Box

To hook the UDF to ANSYS FLUENT, first specify the Number of User-Defined Scalars (e.g., 2) in the User-Defined Scalars dialog box (Figure 6.6.3). As you enter the number of user-defined scalars, the dialog box will expand to show the User-Defined Scalars Options group box. Next, for each scalar you have defined, increment the UDS Index and select the Solution Zones (e.g., all fluid zones) and the name of the function (e.g., my_uds_flux::libudf) from the Flux Function drop-down list, and click OK.

6.6.3 Hooking DEFINE_UDS_UNSTEADY **UDFs**

After you have interpreted (Chapter 4: Interpreting UDFs) or compiled (Chapter 5: Compiling UDFs) your DEFINE_UDS_UNSTEADY UDF, the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the User-Defined Scalars dialog box in ANSYS FLUENT.

i Make sure that you have selected Transient from the Time list in the Solver group box of the General task page.

To hook the UDF to ANSYS FLUENT, first open the User-Defined Scalars dialog box.

Define $\longrightarrow$ User-Defined $\longrightarrow$ Scalars...
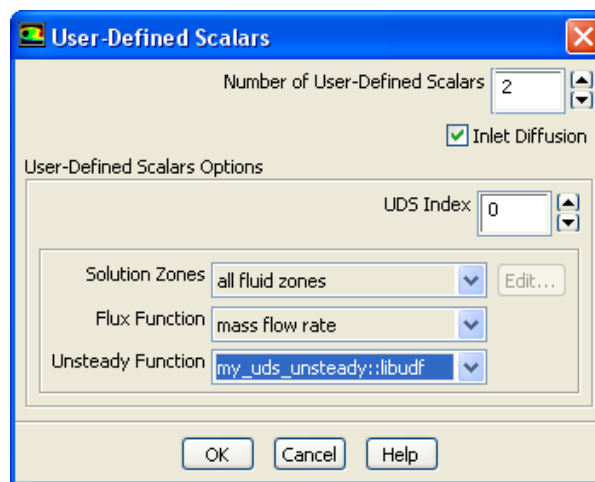
Figure 6.6.4: The User-Defined Scalars Dialog Box

In the User-Defined Scalars dialog box (Figure 6.6.4), specify the Number of User-Defined Scalars (e.g., 2) in the User-Defined Scalars dialog box (Figure 6.6.4). As you enter the number of user-defined scalars, the dialog box will expand to show the User-Defined Scalars Options group box. Next, for each scalar you have defined, increment the UDS Index and select the Zone Type and the Flux Function. Then select the name of your UDF (e.g., my_uds_unsteady::libudf) from the Unsteady Function drop-down list, and click OK.

6.7   Common Errors While Hooking a UDF to ANSYS FLUENT

In some cases, if you select user-defined as an option in a graphics dialog box but have not previously interpreted or compiled/loaded a UDF, you will get an error message.

In other graphics dialog boxes, the user-defined option will only become visible as an option for a parameter *after* you have interpreted or compiled the UDF. After you have interpreted or compiled the UDF, you can then select user-defined option and the list of interpreted and compiled/loaded UDFs will be displayed.

If you inadvertently hook a UDF to the wrong parameter in an ANSYS FLUENT graphics dialog box (e.g., profile UDF for a material property), you will either get a real-time error message, or when you go to initialize or iterate the solution, ANSYS FLUENT will report an error in the dialog box (Figure 6.7.1).
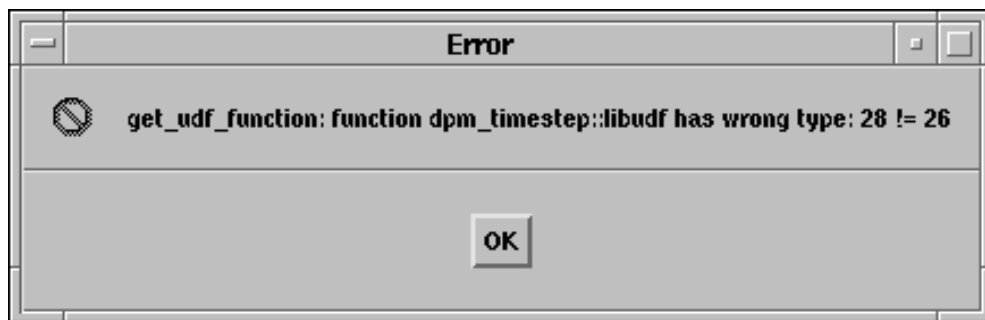
Figure 6.7.1: The Error Dialog

A message will also be reported to the console (and log file):

```
Error: get_udf_function: function dpm_timestep::libudf has wrong type: 28 != 26
Error Object: #f
```

Chapter 7.   Parallel Considerations

This chapter contains an overview of user-defined functions (UDFs) for parallel ANSYS FLUENT and their usage. Details about parallel UDF functionality can be found in the following sections:

- Section 7.1: Overview of Parallel ANSYS FLUENT

- Section 7.2: Cells and Faces in a Partitioned Mesh

- Section 7.3: Parallelizing Your Serial UDF

- Section 7.4: Parallelization of Discrete Phase Model (DPM) UDFs

- Section 7.5: Macros for Parallel UDFs

- Section 7.6: Limitations of Parallel UDFs

- Section 7.7: Process Identification

- Section 7.8: Parallel UDF Example

- Section 7.9: Writing Files in Parallel

7.1 Overview of Parallel ANSYS FLUENT

ANSYS FLUENT's parallel solver computes a solution to a large problem by simultaneously using multiple processes that may be executed on the same machine, or on different machines in a network. It does this by splitting up the computational domain into multiple partitions (Figure 7.1.1) and assigning each data partition to a different compute process, referred to as a compute node (Figure 7.1.2.) Each compute node executes the same program on its own data set, simultaneously, with every other compute node. The host process, or simply the host, does not contain mesh cells, faces, or nodes (except when using the DPM shared-memory model). Its primary purpose is to interpret commands from Cortex (the ANSYS FLUENT process responsible for user-interface and graphics-related functions) and in turn, to pass those commands (and data) to a compute node which distributes it to the other compute nodes.
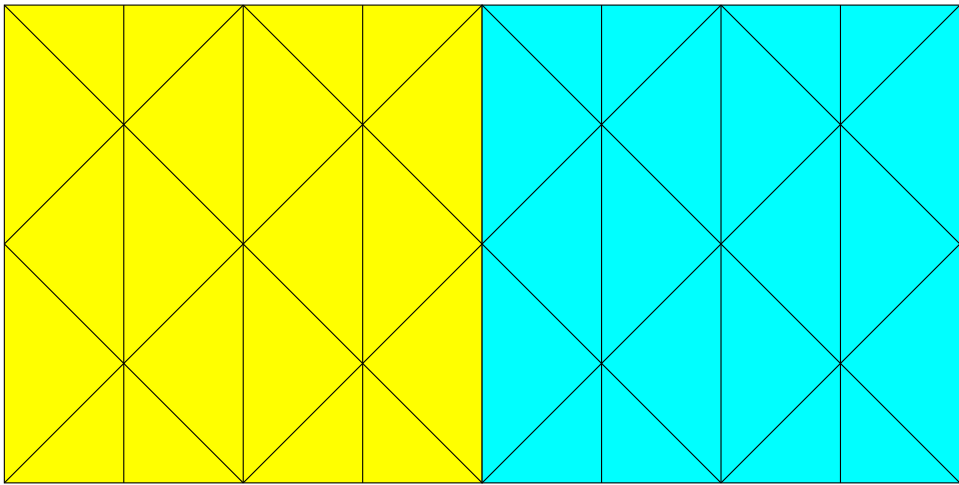
Figure 7.1.1: Partitioned Mesh in Parallel ANSYS FLUENT
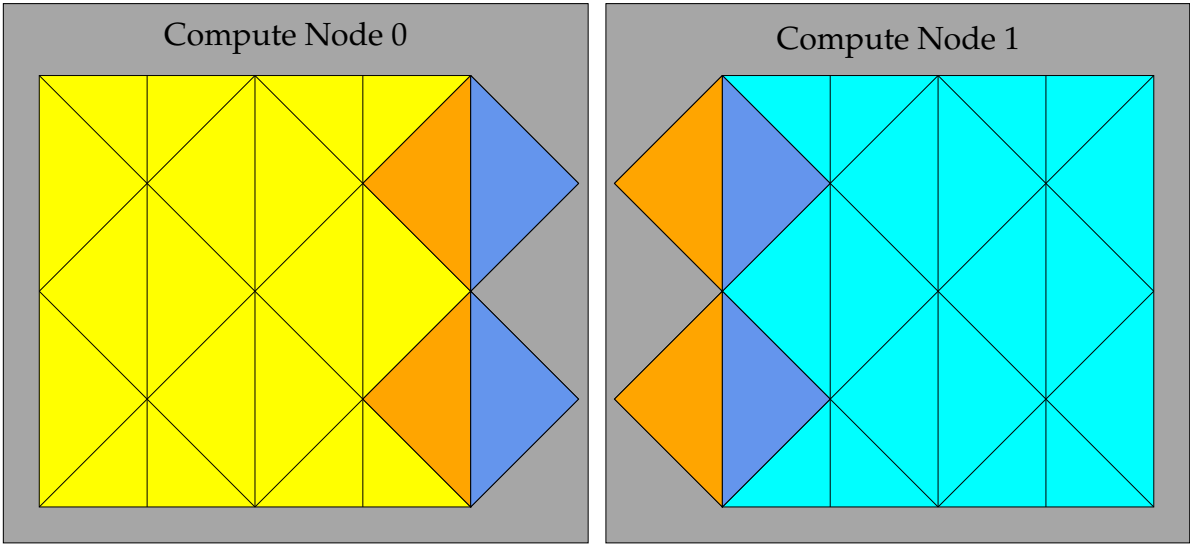
Figure 7.1.2: Partitioned Mesh Distributed Between Two Compute Nodes
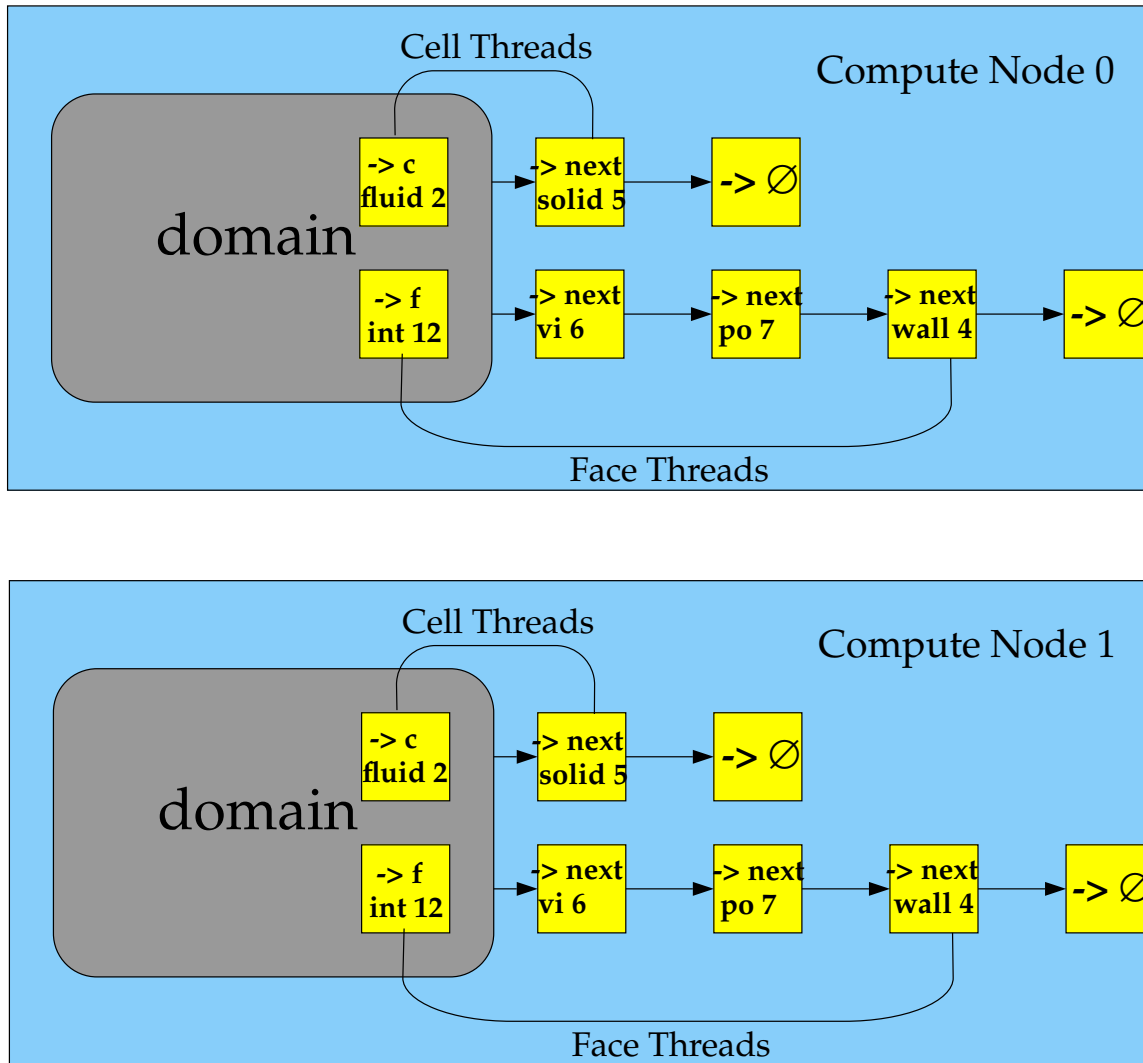
Figure 7.1.3: Domain and Thread Mirroring in a Distributed Mesh

Compute nodes store and perform computations on their portion of the mesh while a single layer of overlapping cells along partition boundaries provides communication and continuity across the partition boundaries (Figure 7.1.2). Even though the cells and faces are partitioned, all of the domains and threads in a mesh are mirrored on each compute node (Figure 7.1.3). The threads are stored as linked lists as in the serial solver. The compute nodes can be implemented on a massively parallel computer, a multiple-CPU workstation, or a network of workstations using the same or different operating systems.

7.1.1 Command Transfer and Communication

The processes that are involved in an ANSYS FLUENT session running in parallel are defined by Cortex, a host process, and a set of n compute node processes (referred to as compute nodes), with compute nodes being labeled from 0 to n-1 (Figure 7.1.4). The host receives commands from Cortex and passes commands to compute node-0. Compute node-0, in turn, sends commands to the other compute nodes. All compute nodes (except 0) receive commands from compute node-0. Before the compute nodes pass messages to the host (via compute node-0), they synchronize with each other. Figure 7.1.4 shows the relationship of processes in parallel ANSYS FLUENT.

Each compute node is "virtually" connected to every other compute node and relies on its "communicator" to perform such functions as sending and receiving arrays, synchronizing, performing global reductions (such as summations over all cells), and establishing machine connectivity. An ANSYS FLUENT communicator is a message-passing library. For example, it could be a vendor implementation of the Message Passing Interface (MPI) standard, as depicted in Figure 7.1.4.

All of the parallel ANSYS FLUENT processes (as well as the serial process) are identified by a unique integer ID. The host process is assigned the ID node_host(=999999). The host collects messages from compute node-0 and performs operation (such as printing, displaying messages, and writing to a file) on all of the data, in the same way as the serial solver. (Figure 7.1.5)
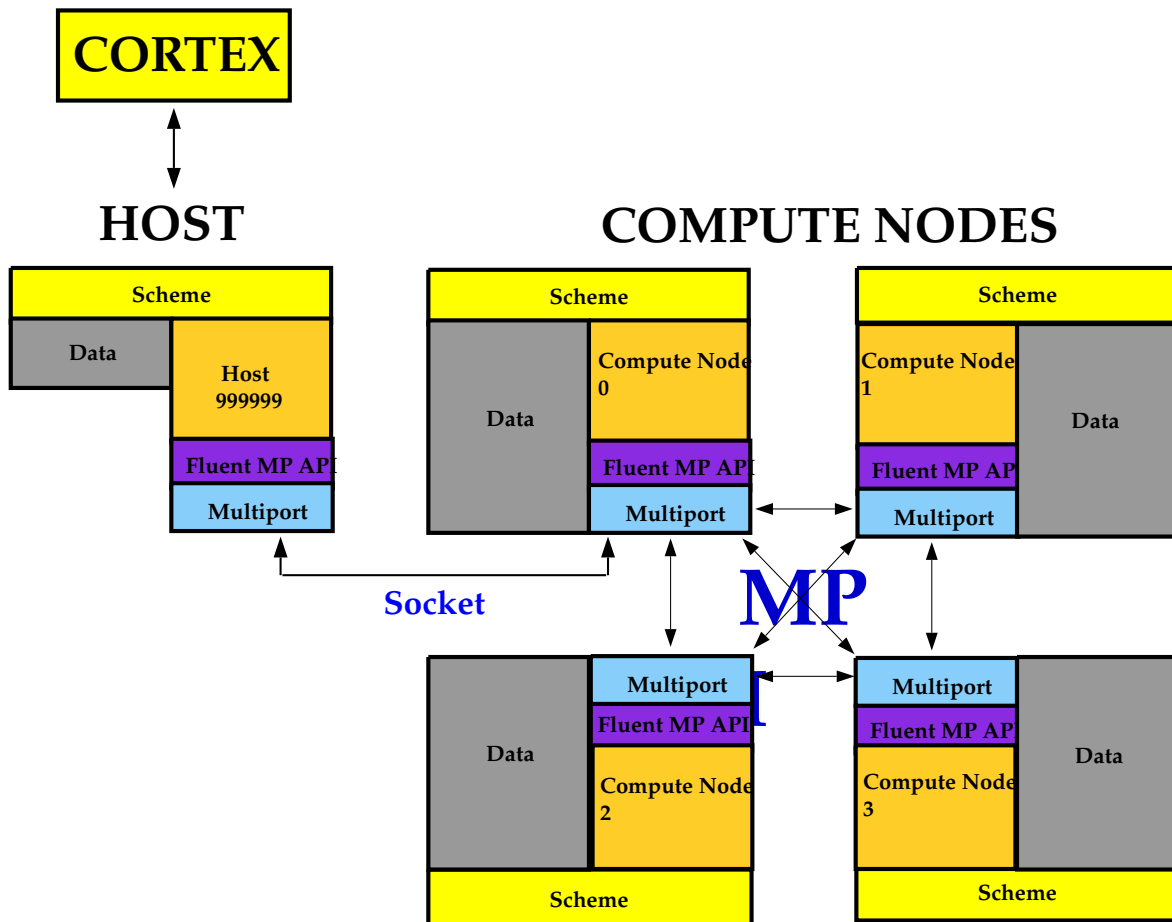
Figure 7.1.4: Parallel ANSYS FLUENT Architecture
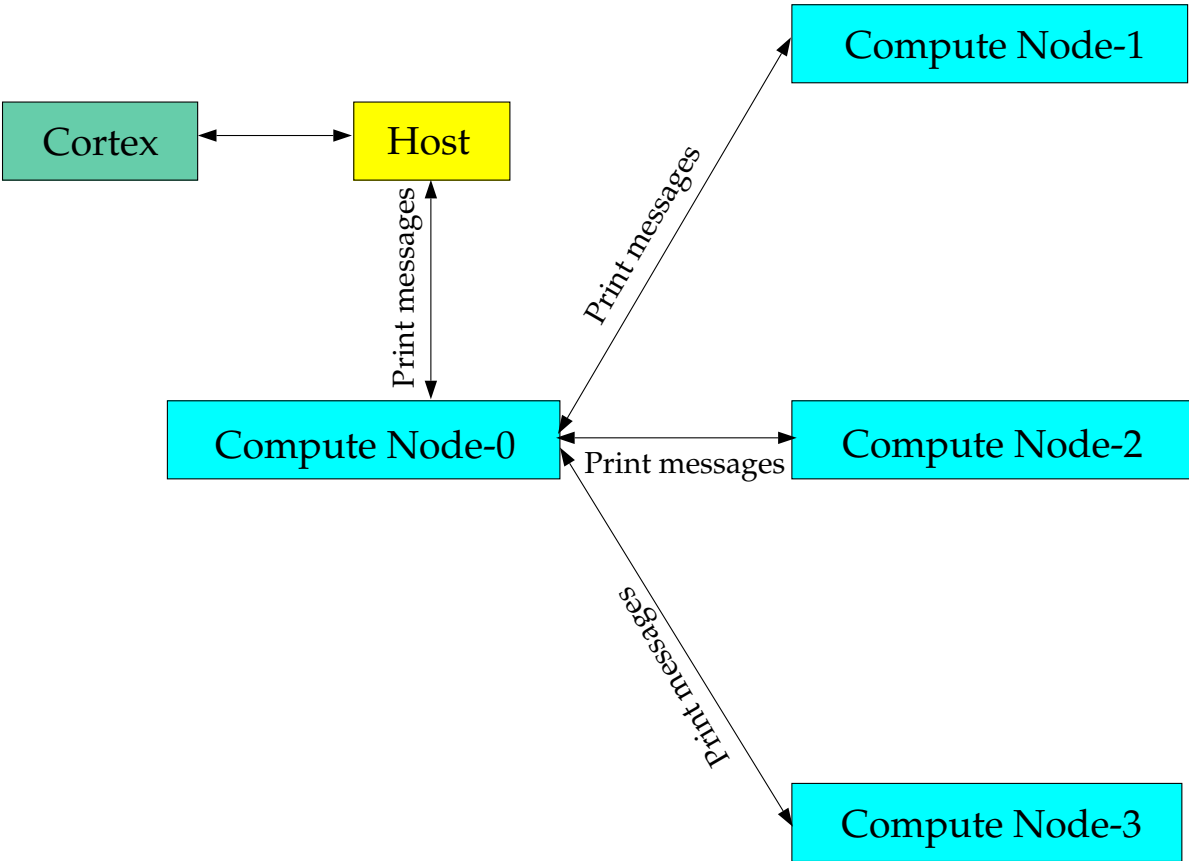
Figure 7.1.5: Example of Command Transfer in Parallel ANSYS FLUENT

7.2 Cells and Faces in a Partitioned Mesh

Some terminology needs to be introduced to distinguish between different types of cells and faces in a partitioned mesh. Note that this nomenclature applies only to parallel coding in ANSYS FLUENT.

Cell Types in a Partitioned Mesh

There are two types of cells in a partitioned mesh: *interior cells* and *exterior cells* (Figure 7.2.1). Interior cells are fully contained within a mesh partition. Exterior cells on one compute node correspond to the same interior cells in the adjacent compute node. (Figure 7.1.2). This duplication of cells at a partition boundary becomes important when you want to loop over cells in a parallel mesh. There are separate macros for looping over interior cells, exterior cells, and all cells. See Section 7.5.5: Looping Macros for details.
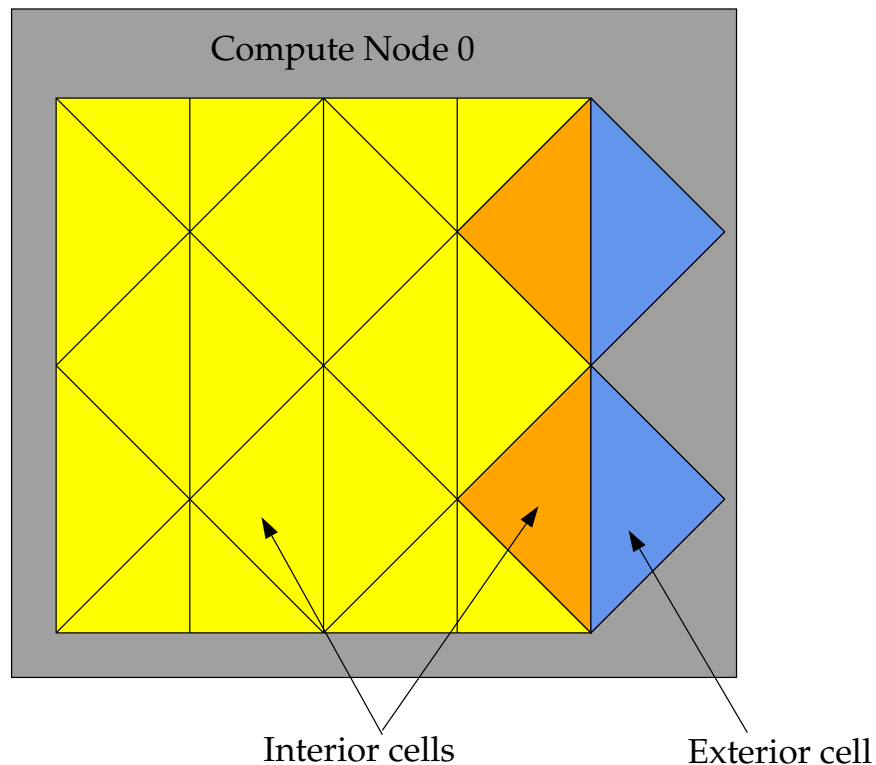
Figure 7.2.1: Partitioned Mesh: Cells

Faces at Partition Boundaries

There are three classifications of faces in a partitioned mesh: *interior*, *boundary zone*, and *external* (Figure 7.2.2). Interior faces have two neighboring cells. Interior faces that lie on a partition boundary are referred to as "partition boundary faces." Boundary zone

faces lie on a physical mesh boundary and have only one adjacent cell neighbor. External faces are non-partition boundary faces that belong to exterior cells. External faces are generally not used in parallel UDFs and, therefore, will not be discussed here.
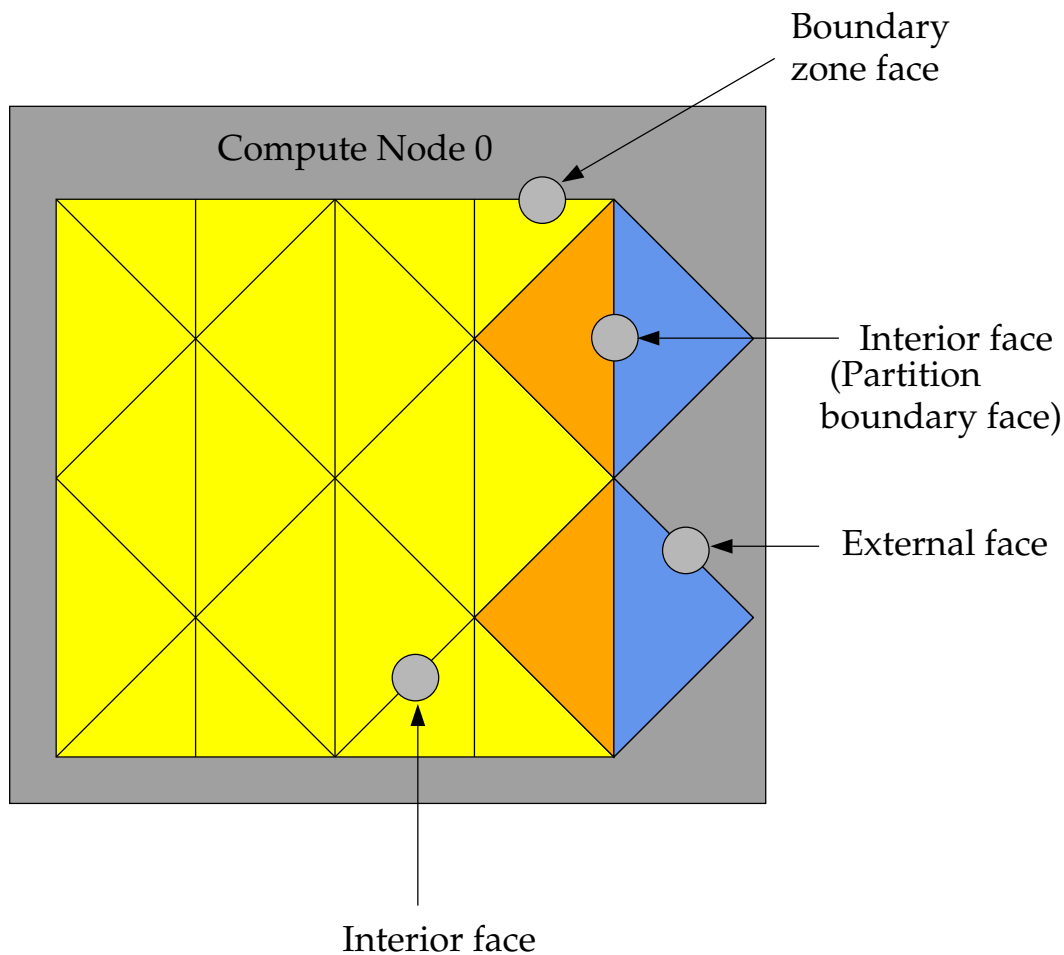
Figure 7.2.2: Partitioned Mesh: Faces

Note that each partition boundary face is duplicated on adjacent compute nodes (Figure 7.1.2). This is necessary so that each compute node can calculate its own face values. However, this duplication can result in face data being counted twice when UDFs are involved in operations that involve summing data in a thread that contains partition boundary faces. For example, if your UDF is tasked with summing data over all of the faces in a mesh, then as each node loops over its faces, duplicated partition boundary faces can be counted twice. For this reason, one compute node in every adjacent set is assigned by ANSYS FLUENT as the "principal" compute node, with respect to partition boundary faces. In other words, although each face can appear on one or two partitions, it can only "officially" belong to one of them. The boolean macro PRINCIPAL_FACE_P(f,t) returns TRUE if the face f is a principal face on the current compute node.

```
PRINCIPAL_FACE_P
```

You can use PRINCIPAL FACE P to test whether a given face is the principal face, before including it in a face loop summation. In the sample source code below, the area of a face is added to the total area only if it is the principal face. Note that PRINCIPAL FACE P is always TRUE for the serial version.

> **ⓘ** PRINCIPAL FACE P can be used *only* in compiled UDFs.

Example

```
begin_f_loop(f,t)
 if PRINCIPAL_FACE_P(f,t)  /* tests if the face is the principle face
                             FOR COMPILED UDFs ONLY */
 {
   F_AREA(area,f,t);   /* computes area of each face  */
   total_area +=NV_MAG(area);  /* computes total face area by
                                   accumulating  magnitude of each
                                   face's area  */
 }
end_f_loop(f,t)
```

Exterior Thread Storage

Each thread stores the data associated with its cells or faces in a set of arrays. For example, pressure is stored in an array and the pressure for cell c is obtained by accessing element c of that array. Storage for exterior cell and face data occurs at the end of every thread data array, as shown in Figure 7.2.3.

7.3   Parallelizing Your Serial UDF

ANSYS FLUENT's serial solver contains Cortex and only a single ANSYS FLUENT process. The parallel solver, on the other hand, contains three types of executable: Cortex, host, and compute node (or simply "node" for short). When ANSYS FLUENT runs in parallel, an instance of Cortex starts, followed by one host and n compute nodes, thereby giving a total of n+2 running processes. For this reason, when you are running in parallel, you will need to make sure that your function will successfully execute as a host and a node process. At first it may appear that you should write three different versions of your UDF: one for serial, host, and node. Good programming practice, however, would suggest that you write a single UDF that, when compiled, can execute on any of the three versions. This process is referred to in this manual as "parallelizing" your serial UDF. You can do this by adding special macros for parallel as well as compiler directives to your UDF, as described below. Compiler directives, (e.g., #if RP NODE, RP HOST, PARALLEL) and their
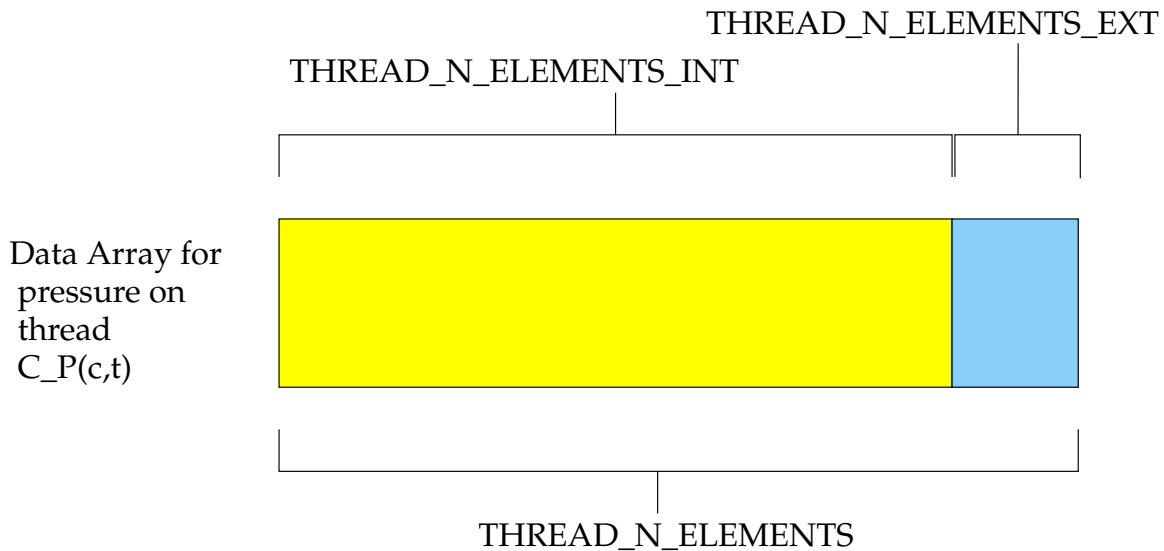
Figure 7.2.3: Exterior Thread Data Storage at End of a Thread Array

negated forms, direct the compiler to include only portions of the function that apply to a particular process, and ignore the rest (see Section 7.5.1: Compiler Directives).

A general rule of thumb is that your serial UDF needs to be "parallelized" if it performs an operation that is dependent on sending or receiving data from another compute node (or the host). UDFs that involve global reductions such as global sums, minimums or maximums, or ones that perform computations on data residing in adjacent compute nodes, for example, will need to be modified in order to run in parallel. Some other types of operations that require parallelization of serial source code include the following:

- Reading and Writing Files

- Global Reductions

- Global Sums

- Global Minimums and Maximums

- Global Logicals

- Certain Loops over Cells and Faces

- Displaying Messages on a Console

- Printing to a Host or Node Process

After the source code for your "parallelized" UDF has been written, it can be compiled using the same methods for serial UDFs. Instructions for compiling UDFs can be found in Chapter 5: Compiling UDFs.

7.4 Parallelization of Discrete Phase Model (DPM) UDFs

The DPM model can be used for the following parallel options:

- Shared Memory

- Message Passing

When you are using a DPM-specific UDF (see Section 2.5: Discrete Phase Model (DPM) DEFINE Macros), it will be executed on the machine that is in charge of the considered particle, based on the above-mentioned parallel options. Since all fluid variables needed for DPM models are held in data structures of the tracked particles, no special care is needed when using DPM UDFs in parallel ANSYS FLUENT with the exception of when you are writing in parallel to a sampling output file. In this case, you are not allowed to use the C function `fprintf`. Instead new functions are provided to enable the parallel file writing. Each node writes its information to separate files, which are put together and sorted upon closure of the file by ANSYS FLUENT. The new functions can be used with the same parameter lists as the C function `fprintf`. The sorting of the files in parallel requires the specification of an extended parameter list. Information can be placed at the top of the file that will not sorted by using the function `par_fprintf_head`:

```
par_fprintf_head("x-coordinate y-coordinate z-coordinate\n")
```

This function will place the string `"x-coordinate y-coordinate z-coordinate"` at the top of the file.

Information is put on the nodes using the function `par_fprintf`:

```
par_fprintf("%d %d %e %e %e\n", P_INJ_ID(P_INJECTION(p)), p->part_id,
P_POS(p)[0], P_POS(p)[1], P_POS(p)[2]);
```

Here, the additional parameters `P_INJ_ID(P_INJECTION(p))` and `p->part_id` are required for the sorting in parallel, but must be included whether you are using the serial or parallel version of ANSYS FLUENT. The output written to the node-specific file of these two parameters will be removed.

An example that utilizes these macros can be found in Section 2.5.8: `DEFINE_DPM_OUTPUT`.

Note that if you need to access other data such as cell values, then for the parallel options except Shared Memory, you will have access to all fluid and solver variables. When you

choose the Shared Memory option, however, you will have access only to the variables defined in the macros SV␣DPM␣LIST and SV␣DPMS␣LIST. These macro definitions can be found in dpm.h.

7.5 Macros for Parallel UDFs

This section contains macros that you can use to parallelize your serial UDF. Where applicable, definitions for these macros can be found in the referenced header file (e.g., para.h).

7.5.1 Compiler Directives

When converting a UDF to run in parallel, some parts of the function may need to be done by the host and some by the compute nodes. This distinction is made when the UDF is compiled. By using ANSYS FLUENT-provided compiler directives, you can specify portions of your function to be assigned to the serial process, the host, or to the compute nodes. The UDF that you write will be written as a single file for the serial, parallel host and parallel node versions, but different parts of the function will be compiled to generate different versions of the dynamically linked shared object file libudf.so (libudf.dll on Windows). Print tasks, for example, may be assigned exclusively to the host, while a task such as computing the total volume of a complete mesh will be assigned to the compute nodes. Since most operations are executed by the serial solver and either the host or compute nodes, negated forms of compiler directives are more commonly used.

Note that the primary purpose of the host is to interpret commands from Cortex and to pass those commands (and data) to compute node-0 for distribution. Since the host does not contain mesh data, you will need to be careful not to include the host in any calculations that could, for example result in a division by zero. In this case, you will need to direct the compiler to ignore the host when it is performing mesh-related calculations, by wrapping those operations around the #if !RP␣HOST directive. For example, suppose that your UDF will compute the total area of a face thread, and then use that total area to compute a flux. If you do not exclude the host from these operations, the total area on the host will be zero and a floating point exception will occur when the function attempts to divide by zero to obtain the flux.

Example

```
#if !RP_HOST
avg_pres = total_pres_a / total_area;  /* if you don't exclude the host
            this operation will result in a division by zero and error!
            Remember that host has no data so its total will be zero.*/
#endif
```

You will need to use the #if !RP␣NODE directive when you want to exclude compute nodes from operations for which they do not have data.

Below is a list of parallel compiler directives and what they do. Note that if either RP␣HOST or RP␣NODE are true, then PARALLEL is also true.

```
/**********************************************************************/
/*    Compiler Directives                                           */
/**********************************************************************/

#if RP_HOST
    /*  only host process is involved */
#endif

#if RP_NODE
    /*  only compute nodes are involved */
#endif

#if PARALLEL
    /* both host and compute nodes are involved, but not serial
        equivalent to #if RP_HOST || RP_NODE  */
#endif


/**********************************************************************/
/*    Negated forms that are more commonly used                    */
/**********************************************************************/

#if !RP_HOST
    /*  either serial or compute node process is involved */
#endif

#if !RP_NODE
    /* either serial or host process is involved */
#endif

#if !PARALLEL
    /*  only serial process is involved */
#endif
```

The following simple UDF shows the use of compiler directives. The adjust function is used to define a function called where␣am␣i. This function queries to determine which type of process is executing and then displays a message on that computed node's monitor.

Example

```
/*******************************************************
   Simple UDF that uses compiler directives
*******************************************************/
#include "udf.h"
DEFINE_ADJUST(where_am_i, domain)
{
#if RP_HOST
  Message("I am in the host process\n");
#endif /* RP_HOST */

#if RP_NODE
  Message("I am in the node process with ID %d\n",myid);
  /* myid is a global variable which is set to the multiport ID for
     each node */
#endif /* RP_NODE */

#if !PARALLEL
  Message("I am in the serial process\n");
#endif /* !PARALLEL */
}
```

This simple allocation of functionality between the different types of processes is useful in a limited number of practical situations. For example, you may want to display a message on the compute nodes when a particular computation is being run (by using RP_NODE or !RP_HOST). Or, you can also choose to designate the host process to display messages (by using RP_HOST or !RP_NODE). Usually you want messages written only once by the host process (and the serial process). Simple messages such as "Running the Adjust Function" are straightforward. Alternatively, you may want to collect data from all the nodes and print the total once, from the host. To perform this type of operation your UDF will need some form of communication between processes. The most common mode of communication is between the host and the node processes.

7.5.2   Communicating Between the Host and Node Processes

There are two sets of similar macros that can be used to send data between the host and the compute nodes: `host_to_node_type_num` and `node_to_host_type_num`.

Host-to-Node Data Transfer

To send data from the host process to *all* the node processes (indirectly via compute node-0) we use macros of the form:

```
host_to_node_type_num(val_1,val_2,...,val_num);
```

where 'num' is the number of variables that will be passed in the argument list and 'type' is the data type of the variables that will be passed. The maximum number of variables that can be passed is 7. Arrays and strings can also be passed from host to nodes, one at a time, as shown in the examples below.

Examples

```
/*  integer and real variables passed from host to nodes  */
host_to_node_int_1(count);
host_to_node_real_7(len1, len2, width1, width2, breadth1, breadth2, vol);


/*  string and array variables passed from host to nodes  */
char wall_name[]="wall-17";
int thread_ids[10] = {1,29,5,32,18,2,55,21,72,14};

host_to_node_string(wall_name,8); /* remember terminating NUL character */
host_to_node_int(thread_ids,10);
```

Note that these `host_to_node` communication macros do not need to be "protected" by compiler directives for parallel UDFs, because all of these macros automatically do the following:

- send the variable value if compiled as the host version

- receive and then set the local variable if compiled as a compute node version

- do nothing in the serial version

The most common use for this set of macros is to pass parameters or boundary conditions from the host to the nodes processes. See the example UDF in Section 7.8: Parallel UDF Example for a demonstration of usage.

Node-to-Host Data Transfer

To send data from compute node-0 to the host process we use macros of the form:

```
node_to_host_type_num(val_1,val_2,...,val_num);
```

where 'num' is the number of variables that will be passed in the argument list and 'type' is the data type of the variables that will be passed. The maximum number of variables that can be passed is 7. Arrays and strings can also be passed from host to nodes, one at a time, as shown in the examples below.

Note that unlike the host_to_node macros which pass data from the host process to *all* of the compute nodes (indirectly via compute node-0), node_to_host macros pass data *only* from compute node-0 to the host.

Examples

```
/*  integer and real variables passed from compute node-0 to host  */
node_to_host_int_1(count);
node_to_host_real_7(len1, len2, width1, width2, breadth1, breadth2, vol);


/*  string and array variables passed from compute node-0 to host  */
char *string;
int string_length;
real vel[ND_ND];

node_to_host_string(string,string_length);
node_to_host_real(vel,ND_ND);
```

node_to_host macros do not need to be protected by compiler directives (e.g., #if RP_NODE) since they automatically do the following:

- send the variable value if the node is compute node-0 and the function is compiled as a node version

- do nothing if the function is compiled as a node version, but the node is not compute node-0

- receive and set variables if the function is compiled as the host version

- do nothing for the serial version

The most common usage for this set of macros is to pass global reduction results from compute node-0 to the host process. In cases where the value that is to be passed is computed by all of the compute nodes, there must be some sort of collection (such as a summation) of the data from all the compute nodes onto compute node-0 before the single collected (summed) value can be sent. Refer to the example UDF in Section 7.8: Parallel UDF Example for a demonstration of usage and Section 7.5.4: Global Reduction Macros for a full list of global reduction operations.

7.5.3 Predicates

There are a number of macros available in parallel ANSYS FLUENT that expand to logical tests. These logical macros, referred to as "predicates", are denoted by the suffix P and can be used as test conditions in your UDF. The following predicates return TRUE if the condition in the parenthesis is met.

```
/*  predicate definitions from para.h header file */

# define MULTIPLE_COMPUTE_NODE_P (compute_node_count > 1)
# define ONE_COMPUTE_NODE_P (compute_node_count == 1)
# define ZERO_COMPUTE_NODE_P (compute_node_count == 0)
```

There are a number of predicates that allow you to test the identity of the node process in your UDF, using the compute node ID. A compute node's ID is stored as the global integer variable myid (see Section 7.7: Process Identification). Each of the macros listed below tests certain conditions of myid for a process. For example, the predicate I_AM_NODE_ZERO_P compares the value of myid with the compute node-0 ID and returns TRUE when they are the same. I_AM_NODE_SAME_P(n), on the other hand, compares the compute node ID that is passed in n with myid. When the two IDs are the same, the function returns TRUE. Node ID predicates are often used in conditional-if statements in UDFs.

```
/*  predicate definitions from para.h header file */

# define I_AM_NODE_HOST_P (myid == node_host)
# define I_AM_NODE_ZERO_P (myid == node_zero)
# define I_AM_NODE_ONE_P (myid == node_one)
# define I_AM_NODE_LAST_P (myid == node_last)
# define I_AM_NODE_SAME_P(n) (myid == (n))
# define I_AM_NODE_LESS_P(n) (myid < (n))
# define I_AM_NODE_MORE_P(n) (myid > (n))
```

Recall that from Section 7.2: Cells and Faces in a Partitioned Mesh, a face may appear in one or two partitions but in order that summation operations don't count it twice, it is officially allocated to only one of the partitions. The tests above are used with the neighboring cell's partition ID to determine if it belongs to the current partition. The convention that is used is that the smaller-numbered compute node is assigned as the "principal" compute node for that face. PRINCIPAL_FACE_P returns TRUE if the face is located on its principal compute node. The macro can be used as a test condition when you want to perform a global sum on faces and some of the faces are partition boundary faces. (The macro returns TRUE for the serial process). Below is the definition of PRINCIPAL_FACE_P from para.h. See Section 7.2: Cells and Faces in a Partitioned Mesh for more information about PRINCIPAL_FACE_P.

```
/*  predicate definitions from para.h header file */
# define PRINCIPAL_FACE_P(f,t) (!TWO_CELL_FACE_P(f,t) || \
      PRINCIPAL_TWO_CELL_FACE_P(f,t))

# define PRINCIPAL_TWO_CELL_FACE_P(f,t) \
    (!(I_AM_NODE_MORE_P(C_PART(F_C0(f,t),THREAD_T0(t))) || \
      I_AM_NODE_MORE_P(C_PART(F_C1(f,t),THREAD_T1(t)))))
```

7.5.4 Global Reduction Macros

Global reduction operations are those that collect data from all of the compute nodes, and reduce the data to a single value, or an array of values. These include operations such as global summations, global maximums and minimums, and global logicals. These macros begin with the prefix PRF_G and are defined in prf.h. Global summation macros are identified by the suffix SUM, global maximums by HIGH, and global minimums by LOW. The suffixes AND and OR identify global logicals.

The variable data types for each macro are identified in the macro name, where R denotes real data types, I denotes integers, and L denotes logicals. For example, the macro PRF_GISUM finds the summation of integers over the compute nodes.

Each of the global reduction macros discussed in the following sections has two different versions: one takes a single variable argument, while the other takes a variable array. Macros with a 1 appended to the end of the name take one argument, and return a single variable as the global reduction result. For example, the macro PRF_GIHIGH1(x) expands to a function that takes one argument x and computes the maximum of the variable x amongst all of the compute nodes, and returns it. The result can then be assigned to another variable (e.g., y), as shown in the following example.

Example: Global Reduction Variable Macro

```
{
 int y;
 int x = myid;
 y = PRF_GIHIGH1(x); /* y now contains the same number (compute_node_count
                             - 1) on all the nodes */
}
```

Macros *without* a 1 suffix, on the other hand, compute global reduction variable arrays. These macros take three arguments: x, N, and iwork where x is an array, N is the number of elements in the array, and iwork is an array that is of the same type and size as x which is needed for temporary storage. Macros of this type are passed an array x and the elements of array x are filled with the new result after returning from the function. For example, the macro PRF_GIHIGH(x,N,iwork) expands to a function that computes the maximum of each element of the array x over all the compute nodes, uses the array iwork for temporary storage, and modifies array x by replacing each element with its resulting global maximum. The function does not return a value.

Example: Global Reduction Variable Array Macro

```
{
 real x[N], iwork[N];

 /* The elements of x are set in the working array here and will
    have different values on each compute node.

    In this case, x[0] could be the maximum cell temperature of all
    the cells on the compute node.  x[1] the maximum pressure, x[2]
    the maximum density, etc.
 */

  PRF_GRHIGH(x,N,iwork); /* The maximum value for each value over
                              all the compute nodes is found here */

  /* The elements of x on each compute node now hold the same
     maximum values over all the compute nodes for temperature,
     pressure, density, etc.  */

}
```

Global Summations

Macros that can be used to compute global sums of variables are identified by the suffix SUM. `PRF_GISUM1` and `PRF_GISUM` compute the global sum of `integer` variables and `integer` variable arrays, respectively.

`PRF_GRSUM1(x)` computes the global sum of a `real` variable `x` across all compute nodes. The global sum is of type `float` when running a single precision version of ANSYS FLUENT and type `double` when running the double precision version. Alternatively, `PRF_GRSUM(x,N,iwork)` computes the global sum of a `float` variable array for single precision and `double` when running double precision.

Global Summations

Macro	Action
`PRF_GISUM1(x)`	Returns sum of integer `x` over all compute nodes.
`PRF_GISUM(x,N,iwork)`	Sets `x` to contain sums over all compute nodes.
`PRF_GRSUM1(x)`	Returns sum of `x` over all compute nodes; `float` if single precision, `double` if double precision.
`PRF_GRSUM(x,N,iwork)`	Sets `x` to contain sums over all compute nodes; `float` array if single precision, `double` array if double precision.

Global Maximums and Minimums

Macros that can be used to compute global maximums and minimums of variables are identified by the suffixes HIGH and LOW, respectively. `PRF_GIHIGH1` and `PRF_GIHIGH` compute the global maximum of `integer` variables and `integer` variable arrays, respectively.

`PRF_GRHIGH1(x)` computes the global maximum of a `real` variable `x` across all compute nodes. The value of the global maximum is of type `float` when running the single precision version of ANSYS FLUENT and type `double` when running the double precision version.

`PRF_GRHIGH(x,N,iwork)` computes the global maximum of a `real` variable array, similar to the description of `PRF_GRSUM(x,N,iwork)` on the previous page. The same naming convention used for `PRF_GHIGH` macros applies to `PRF_GLOW`.

Global Maximums

Macro	Action
PRF_GIHIGH1(x)	Returns maximum of integer x over all compute nodes.
PRF_GIHIGH(x,N,iwork)	Sets x to contain maximums over all compute nodes.
PRF_GRHIGH1(x)	Returns maximums of x over all compute nodes; float if single precision, double if double precision.
PRF_GRHIGH(x,N,iwork)	Sets x to contain maximums over all compute nodes; float array if single precision, double array if double precision.

Global Minimums

Macro	Action
PRF_GILOW1(x)	Returns minimum of integer x over all compute nodes.
PRF_GILOW(x,N,iwork)	Sets x to contain minimums over all compute nodes.
PRF_GRLOW1(x)	Returns minimum of x over all compute nodes; float if single precision, double if double precision.
PRF_GRLOW(x,N,iwork)	Sets x to contain minimums over all compute nodes; float array if single precision, double array if double precision.

Global Logicals

Macros that can be used to compute global logical ANDs and logical ORs are identified by the suffixes AND and OR, respectively. PRF_GLOR1(x) computes the global logical OR of variable x across all compute nodes. PRF_GLOR(x,N,iwork) computes the global logical OR of variable array x. The elements of x are set to TRUE if any of the corresponding elements on the compute nodes are TRUE.

By contrast, PRF_GLAND(x) computes the global logical AND across all compute nodes and PRF_GLAND(x,N,iwork) computes the global logical AND of variable array x. The elements of x are set to TRUE if all of the corresponding elements on the compute nodes are TRUE.

Global Logicals

Macro	Action
PRF_GLOR1(x)	TRUE when variable x is TRUE for *any* of the compute nodes
PRF_GLOR(x,N,work)	TRUE when *any* of the elements in variable array x is TRUE
PRF_GLAND1(x)	TRUE when variable x is TRUE for *all* compute nodes
PRF_GLAND(x,N,iwork)	TRUE when every element in variable array x is TRUE

Global Synchronization

PRF_GSYNC() can be used when you want to globally synchronize compute nodes before proceeding with the next operation. When you insert a PRF_GSYNC macro in your UDF, no commands beyond it will execute until the preceding commands in the source code have been completed on all of the compute nodes. Synchronization may also be useful when debugging your function.

7.5.5 Looping Macros

There are three types of cell looping macros that are available for parallel coding; one that loops over interior cells only, exterior cells only, and both interior and exterior cells.

Looping Over Cells

A partitioned mesh in parallel ANSYS FLUENT is made up of interior cells and exterior cells (see Figure 7.2.1). There is a set of cell-looping macros you can use to loop over interior cells only, exterior cells only, or both interior and exterior cells.

Interior Cell Looping Macro

The macro begin,end_c_loop_int loops over interior cells in a partitioned mesh (Figure 7.5.1) and is identified by the suffix int. This macro pair can also be used by the serial version of ANSYS FLUENT to loop over all cells in the given thread. It contains a begin and end statement, and between these statements, operations can be performed on each of the thread's interior cells in turn. The macro is passed a cell index c and a cell thread pointer tc.

```
begin_c_loop_int(c, tc)
  {
  }
end_c_loop_int(c, tc)
```
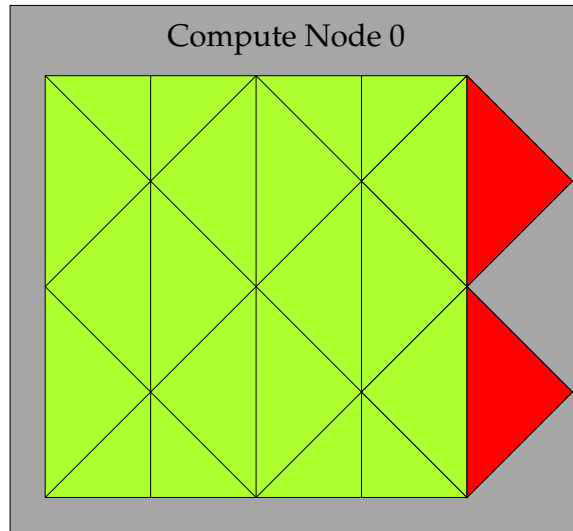
Figure 7.5.1: Looping Over Interior Cells in a Partitioned Mesh Using
begin,end_c_loop_int (indicated by the green cells)

Example

```
real total_volume = 0.0;
begin_c_loop_int(c,tc)
   {
    /*  C_VOLUME gets the cell volume and accumulates it.  The end
        result will be the total  volume of each compute node's
        respective mesh   */
     total_volume += C_VOLUME(c,tc);
   }
end_c_loop_int(c,tc)
```

Exterior Cell Looping Macro

The macro begin,end_c_loop_ext loops over exterior cells in a partitioned mesh (Figure 7.5.2) and is identified by the suffix ext. It contains a begin and end statement, and between these statements, operations can be performed on each of the thread's exterior cells in turn. The macro is passed a cell index c and cell thread pointer tc. In most situations, there is no need to use the exterior cell loop macros. They are only provided for convenience if you come across a special need in your UDF.

```
begin_c_loop_ext(c, tc)
   {
   }
end_c_loop_ext(c,tc)
```
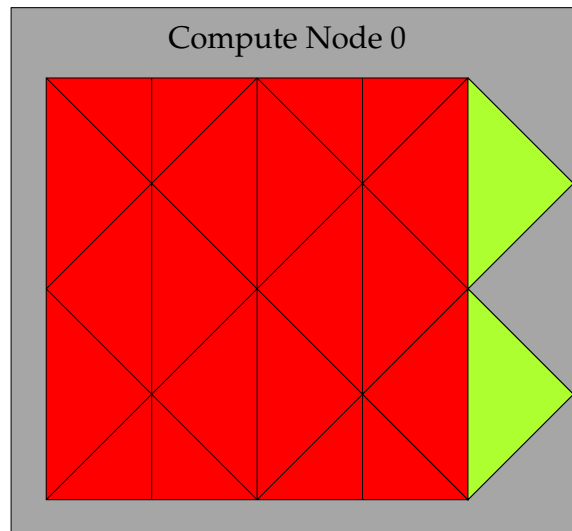

Figure 7.5.2: Looping Over Exterior Cells in a Partitioned Mesh Using
begin,end_c_loop_ext (indicated by the green cells)

Interior and Exterior Cell Looping Macro

The macro begin,end_c_loop can be used in a serial or parallel UDF. In parallel, the
macro will loop over all interior *and* exterior cells in a mesh partition (Figure 7.5.3). Note
that in serial, this pair of macros is equivalent to the begin,end_c_loop_int macros. It
contains a begin and end statement, and between these statements, operations can be
performed on each of the thread's interior and exterior cells in turn. The macro is passed
a cell index c and a cell thread pointer tc.

```
begin_c_loop(c, tc)
   {
   }
end_c_loop(c ,tc)
```
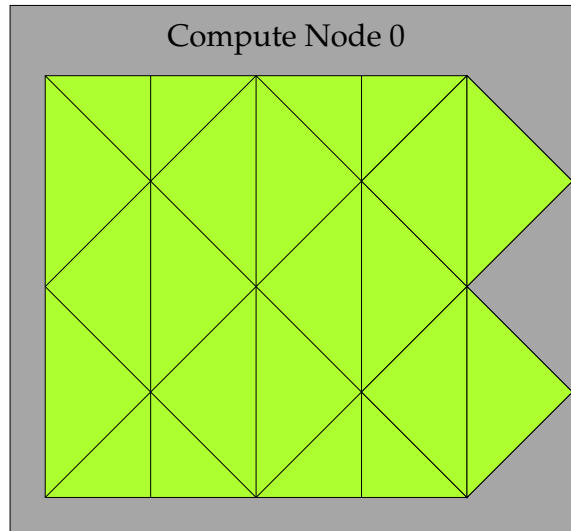
Figure 7.5.3: Looping Over Both Interior and Exterior Cells in a Partitioned Mesh Using `begin,end_c_loop`

Example

```
real temp;
begin_c_loop(c,tc)
   {
     /* get cell temperature, compute temperature function and store
        result in user-defined memory, location index 0.  */

     temp = C_T(c,tc);
     C_UDMI(c,tc,0) = (temp - tmin) / (tmax - tmin);
     /* assumes a valid tmax and tmin has already been computed */
   }
end_c_loop(c,tc)
```

Looping Over Faces

For the purpose of discussing parallel ANSYS FLUENT, faces can be categorized into two types: interior faces and boundary zone faces (Figure 7.2.2). Partition boundary faces are interior faces that lie on the partition boundary of a compute node's mesh.

`begin,end_f_loop` is a face looping macro available in parallel ANSYS FLUENT that loops over all interior and boundary zone faces in a compute node. The macro `begin,end_f_loop` contains a `begin` and `end` statement, and between these statements, operations can be performed on each of the faces of the thread. The macro is passed a face index `f` and face thread pointer `tf`.

```
begin_f_loop(f, tf)
   {
    }
end_f_loop(f,tf)
```

> ℹ️ begin_f_loop_int and begin_f_loop_ext are looping macros that loop
> around interior and exterior faces in a compute node, respectively. The
> _int form is equivalent to begin_f_loop_int. Although these macros exist,
> they do not have a practical application in UDFs and should not be used.

Recall that partition boundary faces lie on the boundary between two adjacent compute
nodes and are represented on both nodes. Therefore, there are some computations (e.g.,
summations) when a partition boundary face will get counted twice in a face loop. This
can be corrected by testing whether the current node is a face's principal compute node
inside your face looping macro, using PRINCIPAL_FACE_P. This is shown in the example
below. See Section 7.2: Cells and Faces in a Partitioned Mesh for details.

Example

```
begin_f_loop(f,tf)
/*  each compute node checks whether or not it is the principal compute
    node with respect to the given face and thread   */

if PRINCIPAL_FACE_P(f,tf)
  /* face is on the principal compute node, so get the area and pressure
     vectors, and compute the total area and pressure for the thread
     from the magnitudes */
{
   F_AREA(area,f,tf);
   total_area += NV_MAG(area);
   total_pres_a += NV_MAG(area)*F_P(f,tf);
 }
end_f_loop(f,tf)

total_area = PRF_GRSUM1(total_area);
total_pres_a = PRF_GRSUM1(total_pres_a);
```

7.5.6   Cell and Face Partition ID Macros

In general, cells and faces have a partition ID that is numbered from 0 to n-1, where n
is the number of compute nodes. The partition IDs of cells and faces are stored in the
variables C_PART and F_PART, respectively. C_PART(c,tc) stores the integer partition ID
of a cell and F_PART(f,tf) stores the integer partition ID of a face.

Note that `myid` can be used in conjunction with the partition ID, since the partition ID of an exterior cell is the ID of the neighboring compute node.

Cell Partition IDs

For interior cells, the partition ID is the same as the compute node ID. For exterior cells, the compute node ID and the partition ID are different. For example, in a parallel system with two compute nodes (0 and 1), the exterior cells of compute node-0 have a partition ID of 1, and the exterior cells of compute node-1 have a partition ID of 0 (Figure 7.5.4).
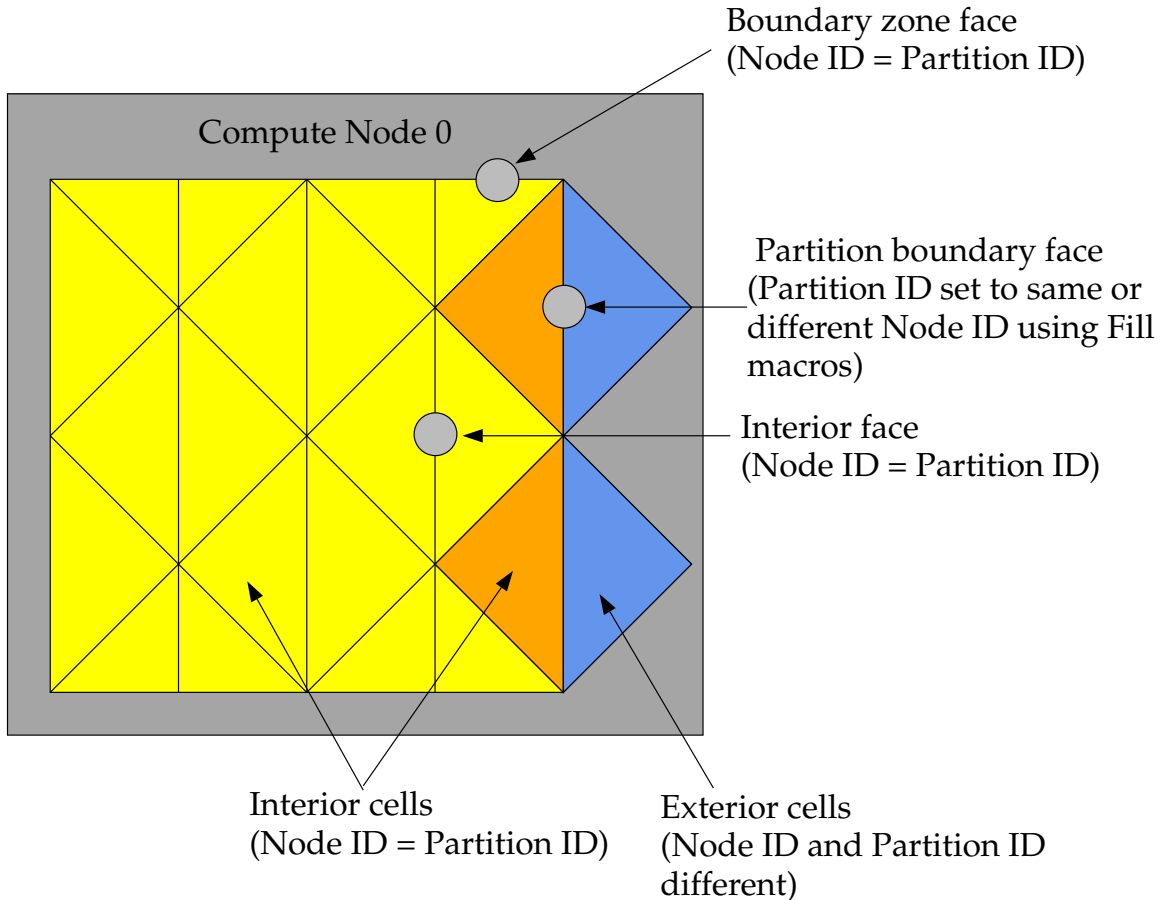
Figure 7.5.4: Partition Ids for Cells and Faces in a Compute Node

Face Partition IDs

For interior faces and boundary zone faces, the partition ID is the same as the compute node ID. The partition ID of a partition boundary face, however, can be either the same as the compute node, or it can be the ID of the adjacent node, depending on what values F_PART is filled with (Figure 7.5.4). Recall that an exterior cell of a compute node has only partition boundary faces; the other faces of the cell belong to the adjacent compute node. Therefore, depending on the computation you want to do with your UDF, you may want to fill the partition boundary face with the same partition ID as the compute node (using Fill_Face_Part_With_Same) or with different IDs (using Fill_Face_Part_With_Different). Face partition IDs will need to be filled before you can access them with the F_PART macro. There is rarely a need for face partition IDs in parallel UDFs.

7.5.7 Message Displaying Macros

You can direct ANSYS FLUENT to display messages on a host, node, or serial process using the Message utility. To do this, simply use a conditional if statement and the appropriate compiler directive (e.g., #if RP_NODE) to select the process(es) you want the message to come from. This is demonstrated in the following example:

Example

```
#if RP_NODE
  Message("Total Area Before Summing %f\n",total\_area);
#endif /* RP_NODE */
```

In this example, the message will be sent by the compute nodes. (It will not be sent by the host or serial process.)

Message0 is a specialized form of the Message utility. Message0 will send messages from compute node-0 only and is ignored on the other compute nodes, without having to use a compiler directive. Note that Message0 will also display messages on a serial process.

Example

```
/* Let Compute Node-0 display messages  */

Message0("Total volume = %f\n",total_volume);
```

7.5.8 Message Passing Macros

High-level communication macros of the form `node_to_host...` and `host_to_node...` that are described in Section 7.5.2: Communicating Between the Host and Node Processes are typically used when you want to send data from the host to all of the compute nodes, or from node-0 to the host. You cannot, however, use these high-level macros when you need to pass data between compute nodes, or pass data from all of the compute nodes to compute node-0. In these cases, you can use special message passing macros described in this section.

Note that the higher-level communication macros expand to functions that perform a number of lower-level message passing operations which send sections of data as single arrays from one process to another process. These lower-level message passing macros can be easily identified in the macro name by the characters `SEND` and `RECV`. Macros that are used to send data to processes have the prefix `PRF_CSEND`, whereas macros that are used to receive data from processes have the prefix `PRF_CRECV`. Data that is to be sent or received can belong to the following data types: character (`CHAR`), integer (`INT`), `REAL` and logical (`BOOLEAN`). `BOOLEAN` variables are `TRUE` or `FALSE`. `REAL` variables are assigned as `float` data types when running a single precision version of ANSYS FLUENTand `double` when running double precision. Message passing macros are defined in the `prf.h` header file and are listed below.

```
/*  message passing macros  */

PRF_CSEND_CHAR(to, buffer, nelem, tag)
PRF_CRECV_CHAR (from, buffer, nelem, tag)
PRF_CSEND_INT(to, buffer, nelem, tag)
PRF_CRECV_INT(from, buffer, nelem, tag)
PRF_CSEND_REAL(to, buffer, nelem, tag)
PRF_CRECV_REAL(from, buffer, nelem, tag)
PRF_CSEND_BOOLEAN(to, buffer, nelem, tag)
PRF_CRECV_BOOLEAN(from, buffer, nelem, tag)
```

There are four arguments to the message passing macros. For 'send' messages, the argument `to` is the node ID of the process that data is being sent to. `buffer` is the name of an array of the appropriate type that will be sent. `nelem` is the number of elements in the array and `tag` is a user-defined message tag. The tag convention is to use `myid` when sending messages.

For 'receive' messages, the argument `from` is the ID of the sending node. `buffer` is the name of an array of the appropriate type that will be received. `nelem` is the number of elements in the array. `tag` is the ID of the sending node, as the convention is to have the `tag` argument the same as the `from` argument (i.e., the first argument) for receive messages.

Note that if variables that are to be sent or received are defined in your function as `real` variables, then you can use the message passing macros with the `_REAL` suffix. The compiler will then substitute `PRF_CSEND_DOUBLE` or `PRF_CRECV_DOUBLE` if you are running double precision and `PRF_CSEND_FLOAT` or `PRF_CRECV_FLOAT`, for single precision.

Because message-passing macros are low-level macros, you will need to make sure that when a message is sent from a node process, a corresponding receiving macro appears in the receiving-node process. Note that your UDF cannot directly send messages from a compute node (other than 0) to the host using message-passing macros. They can send messages indirectly to the host through compute node-0. For example, if you want your parallel UDF to send data from all of the compute nodes to the host for postprocessing purposes, the data will first have to be passed from each compute node to compute node-0, and then from compute node-0 to the host. In the case where the compute node processes send a message to compute node-0, compute node-0 must have a loop to receive the `N` messages from the `N` nodes.

Below is an example of a compiled parallel UDF that utilizes message passing macros `PRF_CSEND` and `PRF_CRECV`. Refer to the comments (*/) in the code, for details about the function.

Example: Message Passing

```
#include "udf.h"
#define WALLID 3

DEFINE_ON_DEMAND(face_p_list)
{
#if !RP_HOST  /* Host will do nothing in this udf. Serial will */
  face_t f;
  Thread *tf;
  Domain *domain;
  real *p_array;
  real x[ND_ND], (*x_array)[ND_ND];
  int n_faces, i, j;

  domain=Get_Domain(1); /* Each Node will be able to access
                                its part of the domain */

  tf=Lookup_Thread(domain, WALLID); /* Get the thread from the domain */

  /* The number of faces of the thread on nodes 1,2... needs to be sent
     to compute node-0 so it knows the size of the arrays to receive
     from each */

  n_faces=THREAD_N_ELEMENTS_INT(tf);
```

```
   /* No need to check for Principal Faces as this UDF
      will be used for boundary zones only */

#if RP_NODE
  if(! I_AM_NODE_ZERO_P) /* Nodes 1,2... send the number of faces */
    {
      PRF_CSEND_INT(node_zero, &n_faces, 1, myid);
    }
#endif
/* Allocating memory for arrays on each node */
  p_array=(real * )malloc(n_faces*sizeof(real));
  x_array=(real (*)[ND_ND])malloc(ND_ND*n_faces*sizeof(real));

  begin_f_loop(f, tf)
        /* Loop over interior faces in the thread, filling p_array
           with face pressure and x_array with centroid   */
    {
      p_array[f] = F_P(f, tf);
      F_CENTROID(x_array[f], f, tf);
    }
  end_f_loop(f, tf)

/* Send data from node 1,2, ... to node 0 */
Message0("\nstart\n");
#if RP_NODE
  if(! I_AM_NODE_ZERO_P) /* Only SEND data from nodes 1,2... */
    {
      PRF_CSEND_REAL(node_zero, p_array, n_faces, myid);
      PRF_CSEND_REAL(node_zero, x_array[0], ND_ND*n_faces, myid);
    }
  else
#endif
    {/* Node-0 and Serial processes have their own data,
         so list it out first */
      Message0("\n\nList of Pressures...\n");
            /* Same as Message() on SERIAL */

      for(j=0; j<n_faces; j++)
          /* n_faces is currently node-0/serial value */
        {
# if RP_3D
         Message0("%12.4e %12.4e %12.4e %12.4e\n",
```

```
              x_array[j][0], x_array[j][1], x_array[j][2], p_array[j]);
# else /* 2D */
          Message0("%12.4e %12.4e %12.4e\n",
            x_array[j][0], x_array[j][1], p_array[j]);
# endif
        }
    }


/* Node-0 must now RECV data from the other nodes and list that too */
#if RP_NODE
  if(I_AM_NODE_ZERO_P)
    {
     compute_node_loop_not_zero(i)
          /* See para.h for definition of this loop */
     {
      PRF_CRECV_INT(i, &n_faces, 1, i);
                    /* n_faces now value for node-i */
      /* Reallocate memory for arrays for node-i */
      p_array=(real *)realloc(p_array, n_faces*sizeof(real));
      x_array=(real(*)[ND_ND])realloc(x_array,ND_ND*n_faces*sizeof(real));

      /* Receive data */
      PRF_CRECV_REAL(i, p_array, n_faces, i);
      PRF_CRECV_REAL(i, x_array[0], ND_ND*n_faces, i);
      for(j=0; j<n_faces; j++)
          {
# if RP_3D
    Message0("%12.4e %12.4e %12.4e %12.4e\n",
            x_array[j][0], x_array[j][1], x_array[j][2], p_array[j]);
# else /* 2D */
    Message0("%12.4e %12.4e %12.4e\n",
            x_array[j][0], x_array[j][1], p_array[j]);
# endif
          }
        }
    }
#endif /* RP_NODE */

  free(p_array); /* Each array has to be freed before function exit */
  free(x_array);

#endif /* ! RP_HOST */
}
```

7.5.9   Macros for Exchanging Data Between Compute Nodes

EXCHANGE_SVAR_MESSAGE and EXCHANGE_SVAR_FACE_MESSAGE can be used to exchange storage variables (SV_....) between compute nodes. EXCHANGE_SVAR_MESSAGE exchanges cell data between compute nodes, while EXCHANGE_SVAR_FACE_MESSAGE exchanges face data. Note that compute nodes are 'virtually' synchronized when an EXCHANGE macro is used; receiving compute nodes wait for data to be sent, before continuing.

```
/*  Compute Node Exchange Macros   */

EXCHANGE_SVAR_FACE_MESSAGE(domain, (SV_P, SV_NULL));
EXCHANGE_SVAR_MESSAGE(domain, (SV_P, SV_NULL));
```

EXCHANGE_SVAR_FACE_MESSAGE() is rarely needed in UDFs. You can exchange multiple storage variables between compute nodes. Storage variable names are separated by commas in the argument list and the list is ended by SV_NULL. For example, EXCHANGE_SVAR_MESSAGE(domain, (SV_P, SV_T, SV_NULL)) is used to exchange cell pressure and temperature variables. You can determine a storage variable name from the header file that contains the variable's definition statement. For example, suppose you want to exchange the cell pressure (C_P) with an adjacent compute node. You can look at the header file that contains the definition of C_P (mem.h) and determine that the storage variable for cell pressure is SV_P. You will need to pass the storage variable to the exchange macro.

7.6   Limitations of Parallel UDFs

The macro PRINCIPAL_FACE_P can be used *only* in compiled UDFs.

PRF_GRSUM1 and similar global reduction macros (Section 7.5.4: Global Reduction Macros cannot be used in DEFINE_SOURCE UDFs in parallel ANSYS FLUENT. As a workaround, you can write a DEFINE_ADJUST UDF that calculates a global sum value in the adjust function, and then save the variable in user-defined memory. You can subsequently retrieve the stored variable from user-defined memory and use it inside a DEFINE_SOURCE UDF. This is demonstrated below.

In the following example, the spark volume is calculated in the DEFINE_ADJUST function and the value is stored in user-defined memory using C_UDMI. The volume is then retrieved from user-defined memory and used in the DEFINE_SOURCE UDF.

```
#include "udf.h"

static real spark_center[ND_ND]={20e-3, 1e-3};
static int fluid_chamber_ID = 2;

DEFINE_ADJUST(adjust, domain)
{
 real vol, xc[ND_ND], dis[ND_ND], radius;
 cell_t c;
 Thread * tc;

 tc = Lookup_Thread(domain, fluid_chamber_ID);

 radius = RP_Get_Real("spark/radius");

 vol = 0;
 begin_c_loop_int (c, tc)
    {
     C_CENTROID(xc, c, tc);
     NV_VV(dis, =, xc, -, spark_center);

     if (NV_MAG(dis) < radius)
        {
         vol += C_VOLUME(c, tc);
        }
    }
 end_c_loop_int (c, tc)
 vol = PRF_GRSUM1(vol);

 begin_c_loop_int (c, tc)
    {
     C_UDMI(c, tc, 1) = vol;
    }
 end_c_loop_int (c, tc)
}

DEFINE_SOURCE(energy_source, c, t, dS, eqn)
{
  #if !RP_HOST
    real xc[ND_ND], dis[ND_ND];
    real source, radius, vol, CA, rpm, start_CA;

    rpm       = RP_Get_Real("dynamesh/in-cyn/crank-rpm");
```

```
    start_CA   = RP_Get_Real("spark/start-ca");

    CA = rpm*CURRENT_TIME*6+RP_Get_Real("dynamesh/in-cyn/crank-start-angle");

    if(CA>=start_CA&&CA<(start_CA+RP_Get_Real("spark/duration")*rpm*6))
      {
          radius = RP_Get_Real("spark/radius");
          vol = C_UDMI(c, t, 1);

          C_CENTROID(xc, c, t);
          NV_VV(dis, =, xc, -, spark_center);
          if (NV_MAG(dis) < radius)
            {
             source  =
             RP_Get_Real("spark/energy")/RP_Get_Real("spark/duration")/vol;
             return source;
             }
          else
            {
             return 0;
             }
      }
    else
      {
       return 0;
      }
  #endif
}
```

i Interpreted UDFs cannot be used while running in parallel with an Infini-
band interconnect. The compiled UDF approach should be used in this
case.

7.7 Process Identification

Each process in parallel ANSYS FLUENT has a unique integer identifier that is stored as
the global variable `myid`. When you use `myid` in your parallel UDF, it will return the
integer ID of the current compute node (including the host). The host process has an ID
of `node_host(=999999)` and is stored as the global variable `node_host`. Compute node-0
has an ID of `0` and is assigned to the global variable `node_zero`. Below is a list of global
variables in parallel ANSYS FLUENT.

Global Variables in Parallel ANSYS FLUENT

```
int node_zero = 0;
int node_host = 999999;
int node_one = 1;
int node_serial = 1000000;

int node_last;  /* returns the id of the last compute node  */
int compute_node_count; /* returns the number of compute nodes */
int myid;  /*  returns the id of the current compute node (and host) */
```

`myid` is commonly used in conditional-if statements in parallel UDF code. Below is some sample code that uses the global variable `myid`. In this example, the total number of faces in a face thread is first computed by accumulation. Then, if `myid` is not compute node-0, the number of faces is passed from all of the compute nodes to compute node-0 using the message passing macro `PRF_CSEND_INT`. (See Section 7.5.8: Message Passing Macros for details on `PRF_CSEND_INT`.)

Example: Usage of myid

```
  int noface=0;
  begin_f_loop(f, tf)     /* loops over faces in a face thread and
                              computes number of faces */
    {
     noface++;
    }
  end_f_loop(f, tf)

/* Pass the number of faces from node 1,2, ... to node 0 */

#if RP_NODE
if(myid!=node_zero)
  {
   PRF_CSEND_INT(node_zero, &noface, 1, myid);
  }
#endif
```

7.8   Parallel UDF Example

The following is an example of a serial UDF that has been parallelized, so that it can run on any version of ANSYS FLUENT(host, node, serial). Explanations for the various changes from the simple serial version are provided in the /* comments */ and discussed below. The UDF, named face_av, is defined using an adjust function, computes a global sum of pressure on a specific face zone, and computes its area average.

Example: Global Summation of Pressure on a Face Zone and its Area Average Computation

```
#include "udf.h"

DEFINE_ADJUST(face_av,domain)
{
/* Variables used by serial, host, node versions */
 int surface_thread_id=0;
 real total_area=0.0;
 real total_force=0.0;

/* "Parallelized" Sections */
#if !RP_HOST   /* Compile this section for computing processes only (serial
                  and node)  since these variables are not available
                  on the host */
 Thread* thread;
 face_t face;
 real area[ND_ND];
#endif /* !RP_HOST */



/* Get the value of the thread ID from a user-defined Scheme variable */
#if !RP_NODE /* SERIAL or HOST */
 surface_thread_id = RP_Get_Integer("pres_av/thread-id");
 Message("\nCalculating on Thread # %d\n",surface_thread_id);
#endif /* !RP_NODE */

 /*  To set up this user Scheme variable in cortex type */
 /* (rp-var-define 'pres_av/thread-id 2 'integer #f) */
 /* After set up you can change it to another thread's ID using : */
 /* (rpsetvar 'pres_av/thread-id 7) */

 /* Send the ID value to all the nodes */
 host_to_node_int_1(surface_thread_id); /* Does nothing in serial */
```

```
#if RP_NODE
 Message("\nNode %d is calculating on thread # %d\n",myid,
            surface_thread_id);
#endif /* RP_NODE */

#if !RP_HOST  /* SERIAL or NODE */
 /* thread is only used on compute processes */
 thread = Lookup_Thread(domain,surface_thread_id);

 begin_f_loop(face,thread)

/* If this is the node to which face "officially" belongs,*/
/* get the area vector and pressure and increment         */
/* the total area and total force values for this node    */
  if (PRINCIPAL_FACE_P(face,thread)) /* Always TRUE in serial version */
  {
   F_AREA(area,face,thread);
   total_area += NV_MAG(area);
   total_force += NV_MAG(area)*F_P(face,thread);
  }
 end_f_loop(face,thread)

 Message("Total Area Before Summing %f\n",total_area);
 Message("Total Normal Force Before Summing %f\n",total_force);

# if RP_NODE /* Perform node synchronized actions here
                Does nothing in Serial */
  total_area = PRF_GRSUM1(total_area);
  total_force = PRF_GRSUM1(total_force);
# endif /* RP_NODE */

#endif /* !RP_HOST */

/* Pass the node's total area and pressure to the Host for averaging  */
 node_to_host_real_2(total_area,total_force); /* Does nothing in SERIAL */

#if !RP_NODE /* SERIAL or HOST */
Message("Total Area After Summing: %f (m2)\n",total_area);
Message("Total Normal Force After Summing %f (N)\n",total_force);
Message("Average pressure on Surface %d is %f (Pa)\n",
                surface_thread_id,(total_force/total_area));
#endif /* !RP_NODE */
}
```

The function begins by initializing the variables `surface_thread_id`, `total_area`, and `total_force` for all processes. This is done because the variables are used by the serial, host, and node processes. The compute nodes use the variables for computation purposes and the host uses them for message-passing and displaying purposes. Next, the preprocessor is directed to compile `thread`, `face`, and `area` variables only on the serial and node versions (and not the host), since faces and threads are only defined in the serial and node versions of ANSYS FLUENT. (Note that in general, the host will ignore these statements since its face and cell data are zero, but it is good programming practice to exclude the host. See Section 7.5: Macros for Parallel UDFs for details on compiler directives.)

Next, a user-defined Scheme variable named `pres_av/thread-id` is obtained by the host (and serial) process using the `RP_Get_Integer` utility (see Section 3.6: Scheme Macros), and is assigned to the variable `surface_thread_id`. (Note that this user-defined Scheme variable was previously set up in Cortex and assigned a value of `2` by typing the text commands shown in the comments.) After a Scheme-based variable is set up for the thread ID, it can be easily changed to another thread ID from the text interface, without the burden of modifying the source code and recompiling the UDF. Since the host communicates with Cortex and the nodes are not aware of Scheme variables, it is essential to direct the compiler to exclude the nodes from compiling them using `#if !RP_NODE`. Failure to do this will result in a compile error.

The `surface_thread_id` is then passed from the host to compute node-0 using the `host_to_node` macro. Compute node-0, in turn, automatically distributes the variable to the other compute nodes. The serial and node processes are directed to loop over all faces in the thread associated with the `surface_thread_id`, using `#if !RP_HOST`, and compute the total area and total force. Since the host does not contain any thread data, it will ignore these statements if you do not direct the compiler, but it is good programming practice to do so. The macro `PRINCIPAL_FACE_P` is used to ensure that faces at partition boundaries are not counted twice (see Section 7.2: Cells and Faces in a Partitioned Mesh). The nodes display the total area and force on the monitors (using the `Message` utility) before the global summation. `PRF_GRSUM1` (Section 7.5.4: Global Reduction Macros) is a global summation macro that is used to compute the total area and force of all the compute nodes. These operations are directed for the compute nodes using `#if RP_NODE`.

7.9 Writing Files in Parallel

Although compute nodes can perform computations on data simultaneously when ANSYS
FLUENT is running in parallel, when data is written to a single, common file, the writing
operations have to be sequential. The file has to be opened and written to by processes
that have access to the desired file system. It is often the case that the compute nodes
are running on a dedicated parallel machine without disk space. This means that all of
the data has to be written from the host process which always runs on a machine with
access to a file system, since it reads and writes the case and data files. This implies that
unlike the example in Section 7.5.8: Message Passing Macros, where data is only passed
to compute node-0 to be collated, data must now be passed from all the compute nodes
to compute node-0, which then passes it on to the host node which writes it to the file.
This process is known as "marshalling".

Thus, file writing in parallel is done in the following stages:

1. The host process opens the file.

2. Compute node-0 sends its data to the host.

3. The other compute nodes send their data to compute node-0.

4. Compute node-0 receives the data from the other compute nodes and sends it to
   the host.

5. The host receives the data sent from *all* the compute nodes and writes it to the file.

6. The host closes the file.

Since the `SERIAL`, `HOST`, and `NODE` processes are performing different tasks, the example
below appears long and utilizes a large number of compiler directives. If, however, as
an exercise you make three copies of this example and in each copy delete the unused
sections for either the `SERIAL`, `HOST` or `NODE` versions, then you will see that it is actually
quite a simple routine.

Example: Writing Data to a Common File on the Host Process's File System

```
/*******************************************************************
   This function will write pressures and positions
    for a fluid zone to a file on the host machine
*******************************************************************/
#include "udf.h"

# define FLUID_ID 2
```

```
DEFINE_ON_DEMAND(pressures_to_file)
{
  /* Different variables are needed on different nodes */
#if !RP_HOST
 Domain *domain=Get_Domain(1);
 Thread *thread;
 cell_t c;
#else
 int i;
#endif

#if !RP_NODE
 FILE *fp = NULL;
 char filename[]="press_out.txt";
#endif

#if PARALLEL
 int size;  /* data passing variables */
 real *array;
 int pe;
#endif

 /* Only Serial and Compute Nodes have data on threads */
#if !RP_HOST
 thread=Lookup_Thread(domain,FLUID_ID);
#endif

#if !RP_NODE /* SERIAL or HOST */
 if ((fp = fopen(filename, "w"))==NULL)
      Message("\n Warning: Unable to open %s for writing\n",filename);
 else
      Message("\nWriting Pressure to %s...",filename);
#endif

 /* UDF Now does 3 different things depending on SERIAL, NODE or HOST */

#if !PARALLEL /* SERIAL */
 begin_c_loop(c,thread)
   fprintf(fp, "%g\n", C_P(c,thread));/* Simply write out pressure data */
 end_c_loop(c,thread)
#endif /* !PARALLEL */

#if RP_NODE
```

```
  /* Each Node loads up its data passing array */
  size=THREAD_N_ELEMENTS_INT(thread);
  array = (real *)malloc(size * sizeof(real));

  begin_c_loop_int(c,thread)
    array[c]= C_P(c,thread);
  end_c_loop_int(c,thread)
    /* Set pe to destination node */
    /* If on node_0 send data to host */
    /* Else send to node_0 because */
    /*   compute nodes connect to node_0 & node_0 to host */
  pe = (I_AM_NODE_ZERO_P) ? node_host : node_zero;

  PRF_CSEND_INT(pe, &size, 1, myid);
  PRF_CSEND_REAL(pe, array, size, myid);

  free(array);/* free array on nodes after data sent */

  /* node_0 now collect data sent by other compute nodes */
  /*   and sends it straight on to the host */
  if (I_AM_NODE_ZERO_P)
    compute_node_loop_not_zero (pe)
    {
      PRF_CRECV_INT(pe, &size, 1, pe);
      array = (real *)malloc(size * sizeof(real));
      PRF_CRECV_REAL(pe, array, size, pe);

      PRF_CSEND_INT(node_host, &size, 1, myid);
      PRF_CSEND_REAL(node_host, array, size, myid);

      free((char *)array);
    }
#endif /* RP_NODE */

#if RP_HOST
 compute_node_loop (pe) /* only acts as a counter in this loop */
    {
      /* Receive data sent by each node and write it out to the file */
      PRF_CRECV_INT(node_zero, &size, 1, node_zero);
      array = (real *)malloc(size * sizeof(real));
      PRF_CRECV_REAL(node_zero, array, size, node_zero);
```

```
     for (i=0; i<size; i++)
       fprintf(fp, "%g\n", array[i]);

     free(array);
   }
#endif /* RP_HOST */


#if !RP_NODE /* SERIAL or HOST */
 fclose(fp); /* Close the file that was only opened if on SERIAL or HOST */
 Message("Done\n");
#endif

}
```

Chapter 8.   Examples

This chapter provides examples of UDFs that range from simple to complex. It begins with a step-by-step process that takes you through the seven basic steps of programming and using a UDF in ANSYS FLUENT. Some examples for commonly used applications are subsequently presented.

- Section 8.1: Step-By-Step UDF Example

- Section 8.2: Detailed UDF Examples

8.1 Step-By-Step UDF Example

The following 7-step process can be used to code a UDF and use it effectively in your ANSYS FLUENT model.

8.1.1 Process Overview

1. Define your problem. (Section 8.1.2: Step 1: Define Your Problem)

2. Create a C source code file. (Section 8.1.3: Step 2: Create a C Source File)

3. Start ANSYS FLUENT and read in (or set up) the case file. (Section 8.1.4: Step 3: Start ANSYS FLUENT and Read (or Set Up) the Case File)

4. Interpret or compile the source file. (Section 8.1.5: Step 4: Interpret or Compile the Source File)

5. Hook the UDF to ANSYS FLUENT. (Section 8.1.6: Step 5: Hook the UDF to ANSYS FLUENT)

6. Run the calculation. (Section 8.1.7: Step 6: Run the Calculation)

7. Analyze the numerical solution and compare it to expected results. (Section 8.1.8: Step 7: Analyze the Numerical Solution and Compare to Expected Results)

To begin the process, you'll need to define the problem you wish to solve using a UDF (Step 1). For example, suppose you want to use a UDF to define a custom boundary profile for your model. You will first need to define the set of mathematical equations that describes the profile.

Next you will need to translate the mathematical equation (conceptual design) into a function written in the C programming language (Step 2). You can do this using any text editor. Save the file with a `.c` suffix (e.g., `udfexample.c`) in your working folder. (See Appendix A for some basic information on C programming.)

After you have written the C function, you are ready to start ANSYS FLUENT and read in (or set up) your case file (Step 3). You will then need to interpret or compile the source code, debug it (Step 4), and then hook the function to ANSYS FLUENT (Step 5). Finally you'll run the calculation (Step 6), analyze the results from your simulation, and compare them to expected results (Step 7). You may loop through this entire process more than once, depending on the results of your analysis. Follow the step-by-step process in the sections below to see how this is done.

8.1.2   Step 1: Define Your Problem

The first step in creating a UDF and using it in your ANSYS FLUENT model involves defining your model equation(s).

Consider the elbow duct illustrated in Figure 8.1.1. The domain has a velocity inlet on the left side, and a pressure outlet at the top of the right side.
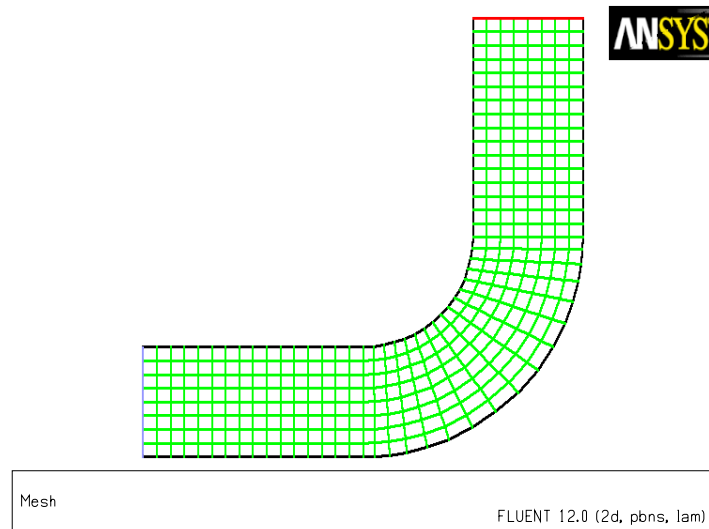
Figure 8.1.1: The Mesh for the Elbow Duct Example

A flow field in which a constant x velocity is applied at the inlet will be compared with one where a parabolic x velocity profile is applied. The results of a constant velocity (of 0.1 m/s) at the inlet are shown in Figures 8.1.2 and 8.1.3.
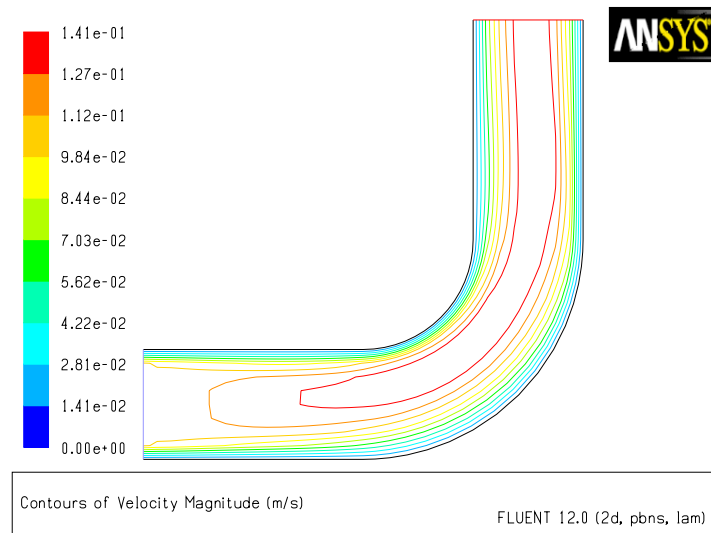
Figure 8.1.2: Velocity Magnitude Contours for a Constant Inlet x Velocity
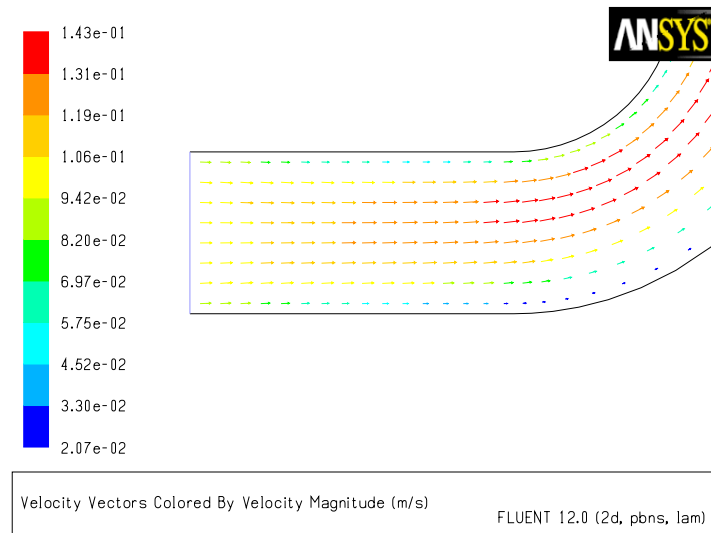
Figure 8.1.3: Velocity Vectors for a Constant Inlet x Velocity

Now suppose that you want to impose a non-uniform x velocity to the duct inlet, which has a parabolic shape. The velocity is 0 m/s at the walls of the inlet and 0.1 m/s at the center.

To solve this type of problem, you can write a custom profile UDF and apply it to your ANSYS FLUENT model.

8.1.3   Step 2: Create a C Source File

Now that you have determined the shape of the velocity profile that defines the UDF, you can use any text editor to create a file containing C code that implements the function. Save the source code file with a `.c` extension (e.g., `myexample.c`) in your working folder. The following UDF source code listing contains only a single function. Your source file can contain multiple concatenated functions. (Refer to Appendix A for basic information on C programming.)

Below is an example of how the profile described in Step 1 can be implemented in a UDF. The functionality of the UDF is designated by the leading `DEFINE` macro. Here, the `DEFINE_PROFILE` macro is used to indicate to the solver that the code that follows will provide profile information at boundaries. Other `DEFINE` macros will be discussed later in this manual. (See Chapter 2: DEFINE Macros for details about `DEFINE` macro usage.)

```
/**************************************************************************
myexample.c
UDF for specifying steady-state velocity profile boundary condition
**************************************************************************/

#include "udf.h"

DEFINE_PROFILE(inlet_x_velocity, thread, position)
{

  real x[ND_ND]; /* this will hold the position vector */
  real y, h;
  face_t f;

  h = 0.016; /* inlet height in m */

  begin_f_loop(f,thread)
  {
    F_CENTROID(x, f, thread);
    y = 2.*(x[1]-0.5*h)/h; /* non-dimensional y coordinate */
    F_PROFILE(f, thread, position) = 0.1*(1.0-y*y);
```

```
  }
  end_f_loop(f, thread)

}
```

The first argument of the DEFINE_PROFILE macro, `inlet_x_velocity`, is the name of
the UDF that you supply. The name will appear in the boundary condition dialog
box after the function is interpreted or compiled, enabling you to hook the function to
your model. Note that the UDF name you supply cannot contain a number as the first
character. The equation that is defined by the function will be applied to all cell faces
(identified by `f` in the face loop) on a given boundary zone (identified by `thread`). The
thread is defined automatically when you hook the UDF to a particular boundary in
the ANSYS FLUENT GUI. The index is defined automatically through the `begin_f_loop`
utility. In this UDF, the `begin_f_loop` macro (Section 3.3: Looping Macros) is used to
loop through all cell faces in the boundary zone. For each face, the coordinates of the
face centroid are accessed by F_CENTROID (Section 3.2.4: Face Centroid (F_CENTROID)).
The y coordinate `y` is used in the parabolic profile equation and the returned velocity is
assigned to the face through F_PROFILE. `begin_f_loop` and F_PROFILE (Section 3.2.6: Set
Boundary Condition Value (F_PROFILE)) are ANSYS FLUENT-supplied macros. Refer to
Chapter 3: Additional Macros for Writing UDFs for details on how to utilize predefined
macros and functions supplied by ANSYS FLUENT to access ANSYS FLUENT solver data
and perform other tasks.

8.1.4   Step 3: Start ANSYS FLUENT and Read (or Set Up) the Case File

After you have created the source code for your UDF, you are ready to begin the problem
setup in ANSYS FLUENT.

1. Start ANSYS FLUENT in Windows using FLUENT Launcher with the following
   settings:

   - Specify the folder that contains your case, data, and UDF source files in the
     Working Directory text box in the General Options tab.

   - If you plan to compile the UDF, make sure that the batch file for the UDF
     compilation environment settings is correctly specified in the UDF Compiler
     tab (see Section 5.1.2: Compilers for further details).

2. Read (or set up) your case file.

8.1.5 Step 4: Interpret or Compile the Source File

You are now ready to interpret or compile the profile UDF (named `inlet_x_velocity`) that you created in Step 2 and is contained within the source file named myexample.c. In general, you *must* compile your function as a compiled UDF if the source code contains structured reference calls or other elements of C that are not handled by the ANSYS FLUENT interpreter. To determine whether you should compile or interpret your UDF, see Section 1.5.1: Differences Between Interpreted and Compiled UDFs.

Interpret the Source File

Follow the procedure below to interpret your source file in ANSYS FLUENT. For more information on interpreting UDFs, see Chapter 4: Interpreting UDFs.

> **i** Note that this step does not apply to Windows parallel networks. See Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box for details.

1. Open the Interpreted UDFs dialog box.

   Define ⟶ User-Defined ⟶ Functions ⟶ Interpreted...
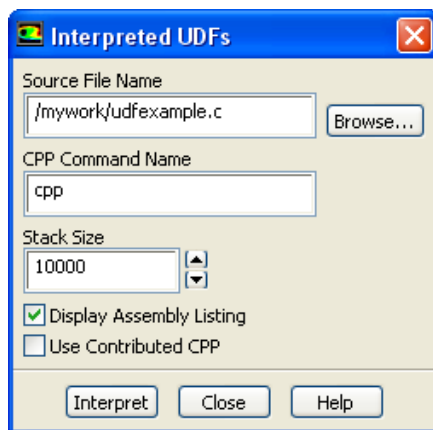
Figure 8.1.4: The Interpreted UDFs Dialog Box

2. In the Interpreted UDFs dialog box, indicate the UDF source file you want to interpret by clicking the Browse... button. This will open the Select File dialog box (Figure 8.1.5).
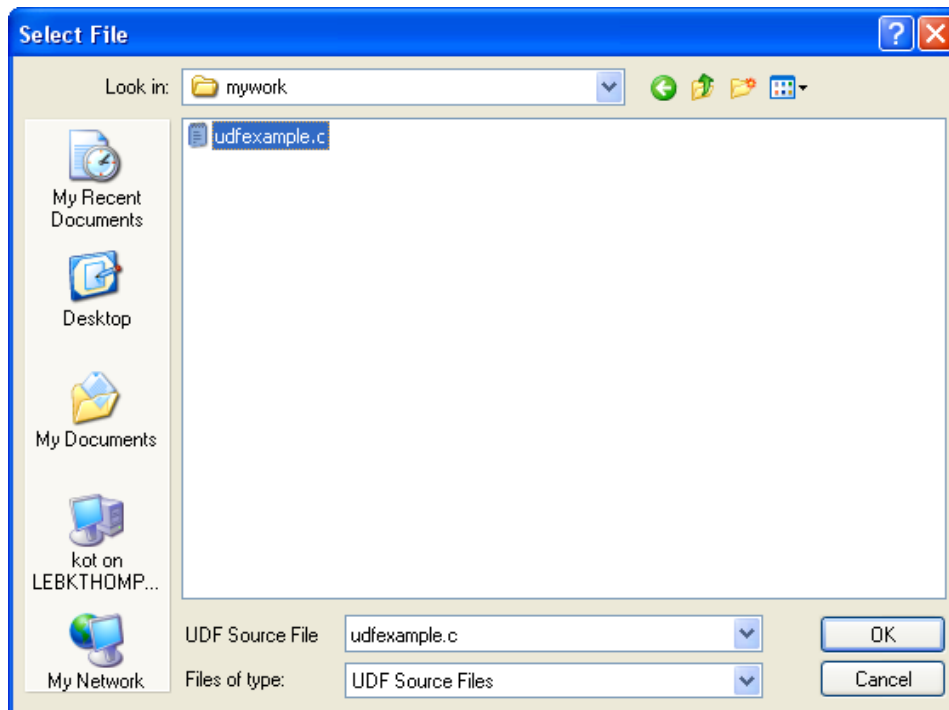
Figure 8.1.5: The Select File Dialog Box

In the Select File dialog box, select the desired file (e.g., myexample.c) and click OK. The Select File dialog box will close and the complete path to the file you selected will appear in the Source File Name text box in the Interpreted UDFs dialog box (Figure 8.1.4).

3. In the Interpreted UDFs dialog box, specify the C preprocessor to be used in the CPP Command Name text box. You can keep the default cpp or you can select Use Contributed CPP to use the preprocessor supplied by ANSYS FLUENT.

4. Keep the default Stack Size setting of 10000, unless the number of local variables in your function will cause the stack to overflow. In this case, set the Stack Size to a number that is greater than the number of local variables used.

5. Enable the Display Assembly Listing option if you want a listing of assembly language code to appear in your console when the function interprets. This option will be saved in your case file, so that when you read the case in a subsequent ANSYS FLUENT session, the assembly code will be automatically displayed.

6. Click Interpret to interpret your UDF. If the Display Assembly Listing option was enabled, then the assembly code will appear in the console when the UDF is interpreted, as shown below.

```
inlet_x_velocity:
                .local.pointer thread (r0)
                .local.int position (r1)
    0           .local.end
    0           save
                .local.int x (r3)
    1           begin.data 8 bytes, 0 bytes initialized:
                .local.float y (r4)
    5           push.float 0
                .local.float h (r5)
    .               .
    .               .
    .               .
  142           pre.inc.int f (r6)
  144           pop.int
  145           b .L3 (28)
      .L2:
  147           restore
  148           restore
  149           ret.v
```

|i| Note that if your compilation is unsuccessful, then ANSYS FLUENT will report an error and you will need to debug your program. See Section 4.3: Common Errors Made While Interpreting A Source File for details.

7. Click Close when the interpreter has finished.

8. Write the case file. The interpreted UDF will be saved with the case file so that the function will be automatically interpreted whenever the case is subsequently read.

Compile the Source File

You can compile your UDF using the text user interface (TUI) or the graphical user interface (GUI) in ANSYS FLUENT. The GUI option for compiling a source file on a Windows system is discussed below. For details about compiling on other platforms, using the TUI to compile your function, or for general questions about compiling UDFs in ANSYS FLUENT see Chapter 5: Compiling UDFs.

1. As mentioned previously make sure that you have started ANSYS FLUENT in Windows using FLUENT Launcher with the following settings:

   - Specify the folder that contains your case, data, and UDF source files in the Working Directory text box in the General Options tab.

   - Make sure that the batch file for the UDF compilation environment settings is correctly specified in the UDF Compiler tab (see Section 5.1.2: Compilers for further details).

2. Open the Compiled UDFs dialog box (Figure 8.1.6).

   Define ⟶ User-Defined ⟶ Functions ⟶ Compiled...
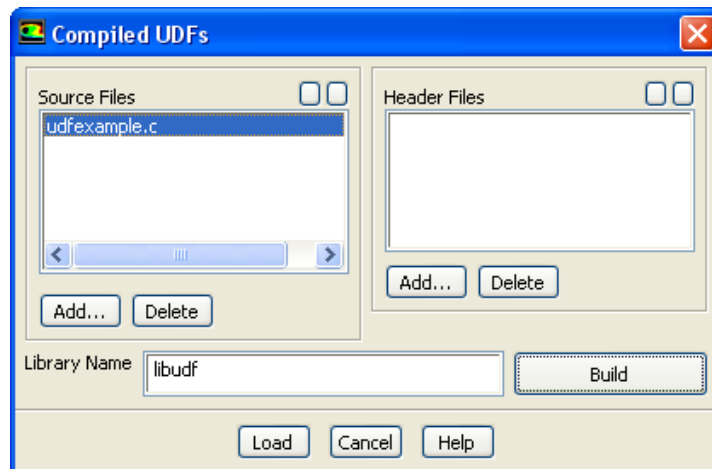
Figure 8.1.6: The Compiled UDFs Dialog Box

3. Click Add... under Source Files in the Compiled UDFs dialog box. This will open the Select File dialog box (Figure 8.1.7).
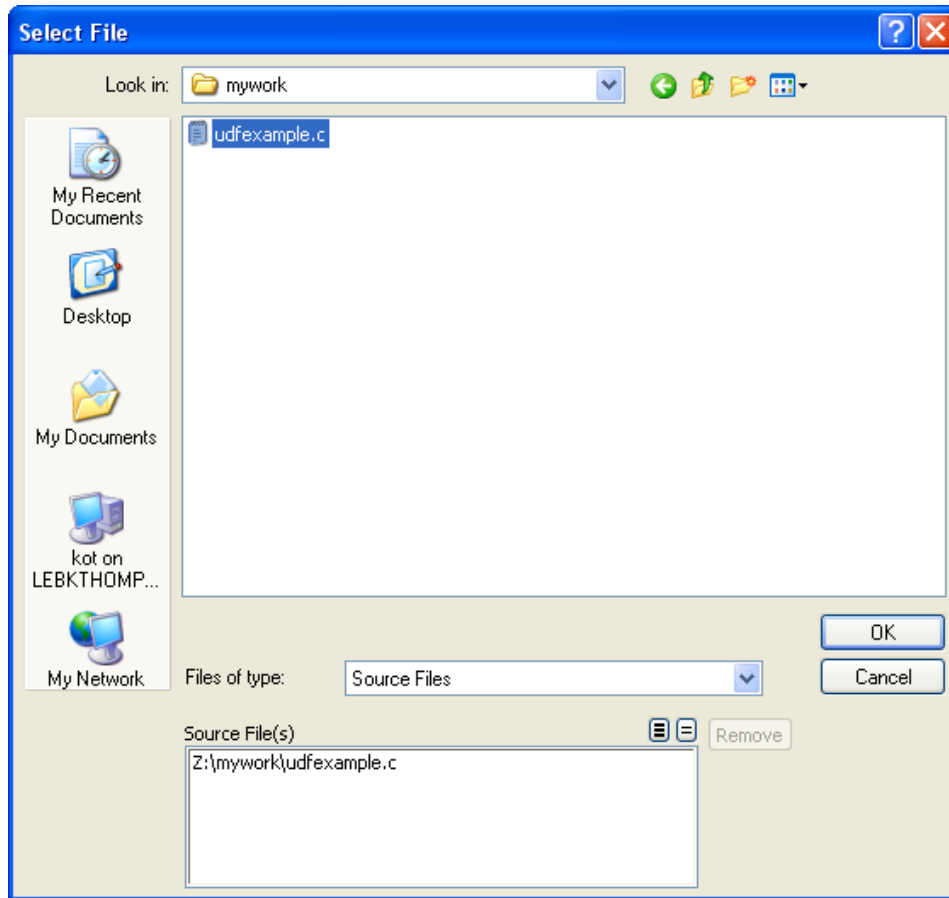
Figure 8.1.7: The Select File Dialog Box

In the Select File dialog box, select the desired file (e.g., udfexample.c) you want to compile. The complete path to the source file will then be displayed under Source File(s). Click OK. The Select File dialog box will close and the file you added will appear in the Source Files list in the Compiled UDFs dialog box.

In a similar manner, select the Header Files that need to be included in the compilation.

4. In the Compiled UDFs dialog box, type the name of the shared library in the Library Name field (or leave the default name libudf). Click Build. This process will compile the code and will build a shared library in your working folder for the architecture you are running on.

As the compile/build process begins, a Warning dialog box will appear, reminding you that the UDF source file must be in the folder that contains your case and data

files (i.e., your working folder). If you have an existing library folder (e.g., libudf) then you will need to remove it prior to the build, to ensure that the latest files are used. Click OK to close the dialog box and resume the compile/build process. The results of the build will be displayed in the console. You can view the compilation history in the log file that is saved in your working folder.

> *i* If the compile/build is unsuccessful, then ANSYS FLUENT will report an error and you will need to debug your program before continuing. See Section 5.6: Common Errors When Building and Loading a UDF Library for a list of common errors.

5. Click Load to load the shared library into ANSYS FLUENT. The console will report that the library has been opened and the function (e.g., `inlet_x_velocity`) loaded.

```
Opening library "libudf"...
Library "libudf\ntx86\2d\libudf.dll" opened
        inlet_x_velocity
Done.
```

See Chapter 5: Compiling UDFs for more information on the compile/build process.

8.1.6 Step 5: Hook the UDF to ANSYS FLUENT

Now that you have interpreted or compiled your UDF following the methods outlined in Step 4, you are ready to hook the profile UDF in this sample problem to the Velocity Inlet boundary condition dialog box (see Chapter 6: Hooking UDFs to ANSYS FLUENT for details on how to hook UDFs). First click the Momentum tab in the Velocity Inlet dialog box (Figure 8.1.8) and then choose the name of the UDF that was given in our sample problem with udf preceding it (udf inlet_x_velocity) from the X-Velocity drop-down list. Click OK to accept the new boundary condition and close the dialog box. The user profile will be used in the subsequent solution calculation.
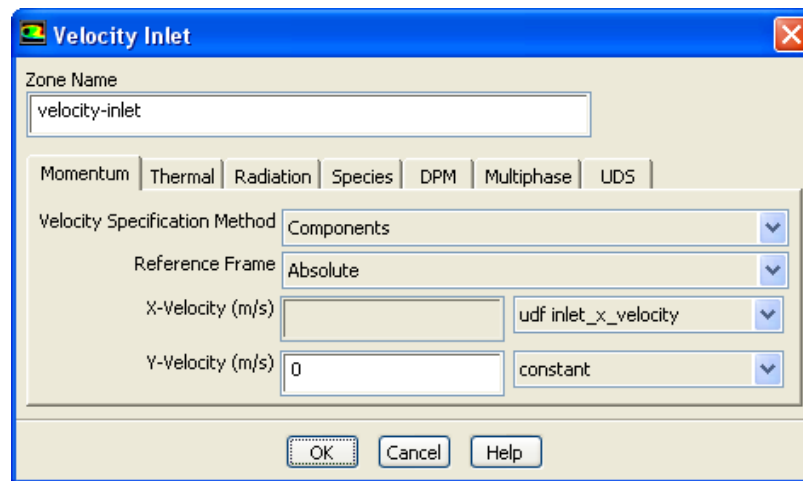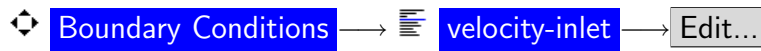
Figure 8.1.8: The Velocity Inlet Dialog Box

8.1.7 Step 6: Run the Calculation

After initializing the solution, run the calculation as usual.

✧ Run Calculation

8.1.8 Step 7: Analyze the Numerical Solution and Compare to Expected Results

After the solution is run to convergence, obtain a revised velocity field. The velocity magnitude contours for the parabolic inlet x velocity are shown in Figure 8.1.9, and can be compared to the results of a constant velocity of 0.1 m/s (Figure 8.1.2). For the constant velocity condition, the constant profile is seen to develop as the flow passes through the duct. The velocity field for the imposed parabolic profile, however, shows a maximum at the center of the inlet, which drops to zero at the walls.
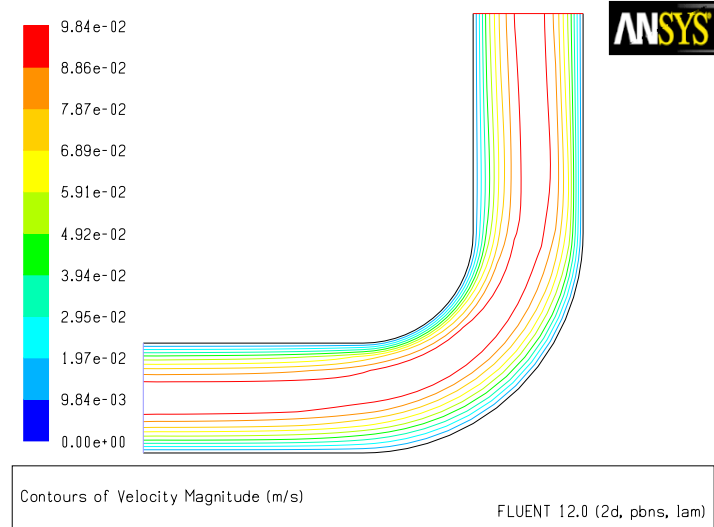
Figure 8.1.9: Velocity Magnitude Contours for a Parabolic Inlet Velocity Profile

8.2   Detailed UDF Examples

This section contains detailed examples of UDFs that are used in typical ANSYS FLUENT applications.

- Section 8.2.1: Boundary Conditions

- Section 8.2.2: Source Terms

- Section 8.2.3: Physical Properties

- Section 8.2.4: Reaction Rates

- Section 8.2.5: User-Defined Scalars

- Section 8.2.6: User-Defined Real Gas Models

8.2.1 Boundary Conditions

This section contains two applications of boundary condition UDFs.

- Parabolic Velocity Inlet Profile for an Elbow Duct

- Transient Velocity Outlet Profile for Flow in a Tube

Parabolic Velocity Inlet Profile in an Elbow Duct

Consider the elbow duct illustrated in Figure 8.2.1. The domain has a velocity inlet on the left side, and a pressure outlet at the top of the right side.
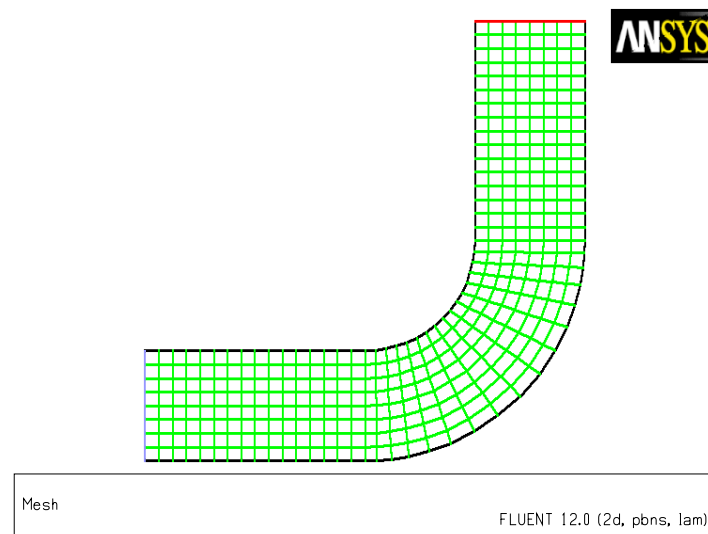
Figure 8.2.1: The Mesh for the Turbine Vane Example

A flow field in which a constant x velocity is applied at the inlet will be compared with one where a parabolic x velocity profile is applied. While the application of a profile using a piecewise-linear profile is available with the boundary profiles option, the specification of a polynomial can only be accomplished by a user-defined function.

The results of a constant velocity (of .01 m/sec) at the inlet are shown in Figures 8.2.2 and 8.2.3. The consistent profile is seen to develop as the flow passes through the duct.
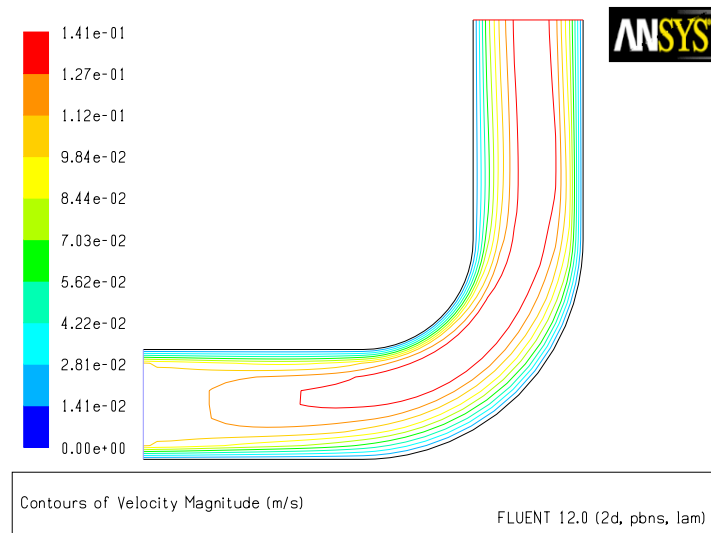
Figure 8.2.2: Velocity Magnitude Contours for a Constant Inlet x Velocity
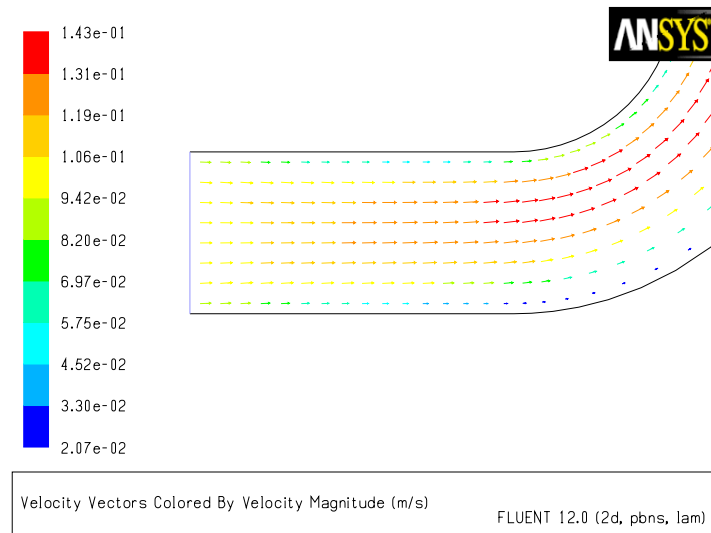
Figure 8.2.3: Velocity Vectors for a Constant Inlet x Velocity

Now suppose that you want to impose a non-uniform x velocity to the duct inlet, which has a parabolic shape. The velocity is 0 m/s at the walls of the inlet and 0.1 m/s at the center.

A UDF is used to introduce this parabolic profile at the inlet. The C source code (`vprofile.c`) is shown below. The function makes use of ANSYS FLUENT-supplied solver functions that are described in Section 3.2.4: Face Macros.

The UDF, named `inlet_x_velocity`, is defined using `DEFINE_PROFILE` and has two arguments: `thread` and `position`. Thread is a pointer to the face's thread, and `position` is an integer that is a numerical label for the variable being set within each loop.

The function begins by declaring variable `f` as a `face_t` data type. A one-dimensional array `x` and variable `y` are declared as `real` data types. A looping macro is then used to loop over each face in the zone to create a profile, or an array of data. Within each loop, `F_CENTROID` outputs the value of the face centroid (array `x`) for the face with index `f` that is on the thread pointed to by `thread`. The y coordinate stored in `x[1]` is assigned to variable `y`, and is then used to calculate the x velocity. This value is then assigned to `F_PROFILE`, which uses the integer `position` (passed to it by the solver based on your selection of the UDF as the boundary condition for x velocity in the Velocity Inlet dialog box) to set the x velocity face value in memory.

```
/*******************************************************************************
vprofile.c
UDF for specifying steady-state velocity profile boundary condition
*******************************************************************************/

#include "udf.h"

DEFINE_PROFILE(inlet_x_velocity, thread, position)
{
  real x[ND_ND]; /* this will hold the position vector */
  real y, h;
  face_t f;

  h = 0.016; /* inlet height in m */

  begin_f_loop(f,thread)
  {
    F_CENTROID(x, f, thread);
    y = 2.*(x[1]-0.5*h)/h; /* non-dimensional y coordinate */
    F_PROFILE(f, thread, position) = 0.1*(1.0-y*y);
  }
  end_f_loop(f, thread)
}
```

To make use of this UDF in ANSYS FLUENT, you will first need to interpret (or compile) the function, and then hook it to ANSYS FLUENT using the graphical user interface. Follow the procedure for interpreting source files using the Interpreted UDFs dialog box (Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box), or compiling source files using the Compiled UDFs dialog box (Section 5.2: Compiling a UDF Using the GUI).

To hook the UDF to ANSYS FLUENT as the velocity boundary condition for the zone of choice, open the Velocity Inlet dialog box and click the Momentum tab (Figure 8.2.4).

Boundary Conditions ⟶ velocity-inlet ⟶ Edit...
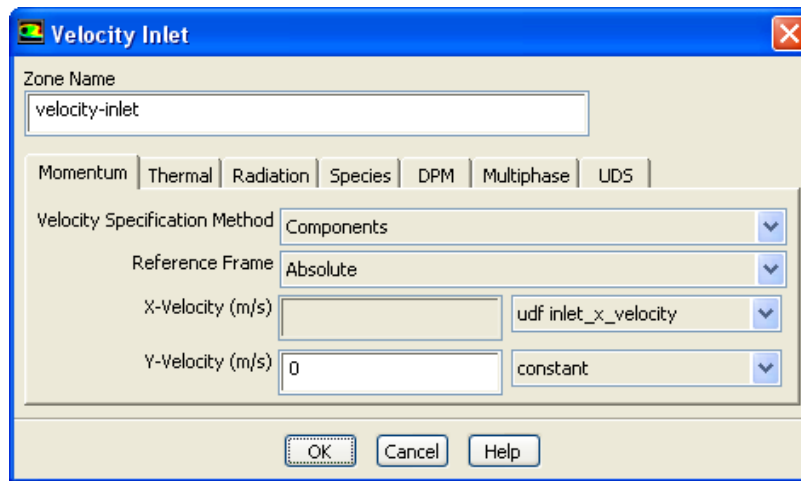
Figure 8.2.4: The Velocity Inlet Dialog Box

In the X-Velocity drop-down list, select udf inlet_x_velocity, the name that was given to the function above (with udf preceding it). Click OK to accept the new boundary condition and close the dialog box. The user profile will be used in the subsequent solution calculation.

After the solution is initialized and run to convergence, a revised velocity field is obtained as shown in Figures 8.2.5 and 8.2.6. The velocity field shows a maximum at the center of the inlet, which drops to zero at the walls.
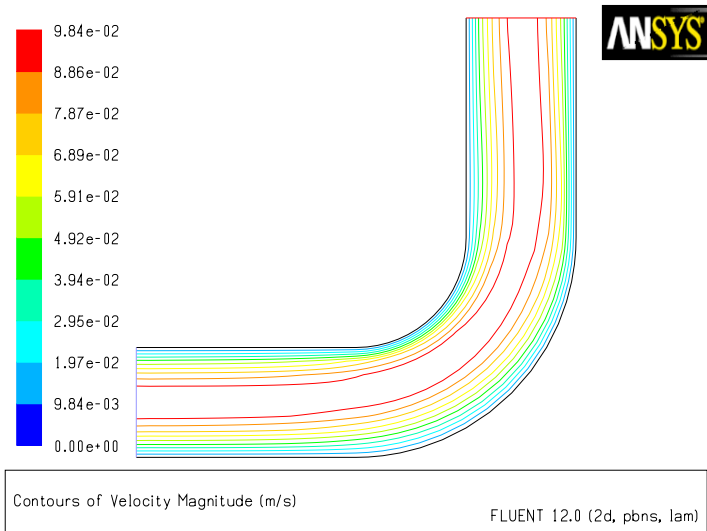
Figure 8.2.5: Velocity Magnitude Contours for a Parabolic Inlet x Velocity
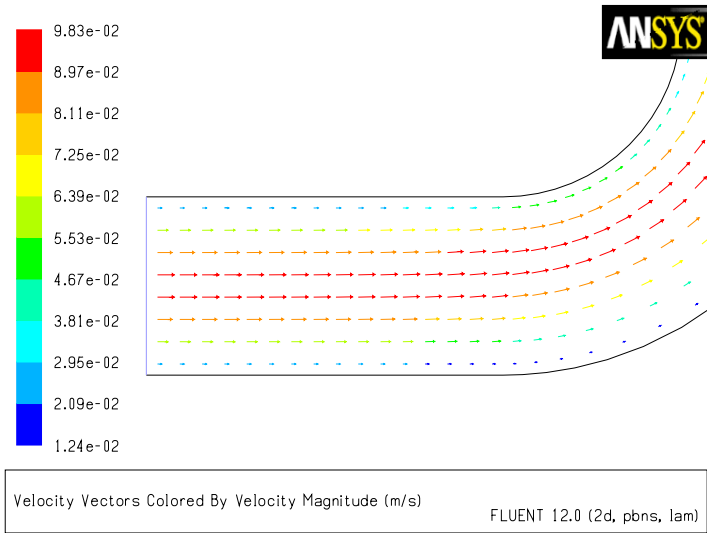
Figure 8.2.6: Velocity Vectors for a Parabolic Inlet x Velocity

Transient Pressure Outlet Profile for Flow in a Tube

In this example, a temporally periodic pressure boundary condition will be applied to the outlet of a tube using a UDF. The pressure has the form

$$p_x = p_0 + A\sin(\omega t)$$

The tube is assumed to be filled with air, with a fixed total pressure at the inlet. The pressure of the air fluctuates at the outlet about an equilibrium value (p_0) of 101325 Pa, with an amplitude of 5 Pa and a frequency of 10 rad/s.

The source file listing for the UDF that describes the transient outlet profile is shown below. The function, named `unsteady_pressure`, is defined using the `DEFINE_PROFILE` macro. The utility `CURRENT_TIME` is used to look up the `real` flow time, which is assigned to the variable `t`. (See Section 3.5: Time-Dependent Macros for details on `CURRENT_TIME`).

```
/**************************************************************************
unsteady.c
UDF for specifying a transient pressure profile boundary condition
**************************************************************************/

#include "udf.h"

DEFINE_PROFILE(unsteady_pressure, thread, position)
{
  face_t f;
  real t = CURRENT_TIME;

  begin_f_loop(f, thread)
  {
    F_PROFILE(f, thread, position) = 101325.0 + 5.0*sin(10.*t);
  }
  end_f_loop(f, thread)

}
```

Before you can interpret or compile the UDF, you must specify a transient flow calculation in the General task page. Then, follow the procedure for interpreting source files using the Interpreted UDFs dialog box (Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box), or compiling source files using the Compiled UDFs dialog box (Section 5.2: Compiling a UDF Using the GUI).

The sinusoidal pressure boundary condition defined by the UDF can now be hooked to the outlet zone. In the Pressure Outlet dialog box (Figure 8.2.7), simply select the name of the UDF given in this example with the word udf preceding it (udf unsteady_pressure) from the Gauge Pressure drop-down list. Click OK to accept the new boundary condition and close the dialog box. The user-defined profile will be used in the subsequent solution calculation.

✧ Boundary Conditions ⟶ ▤ pressure-outlet-5 ⟶ Edit...
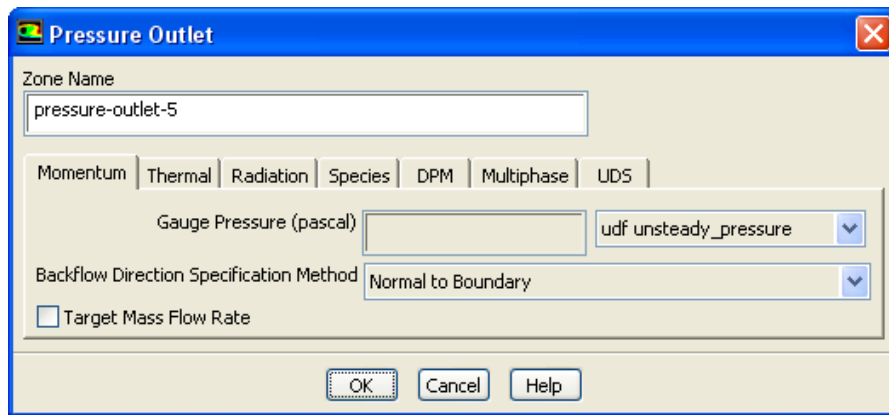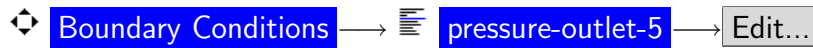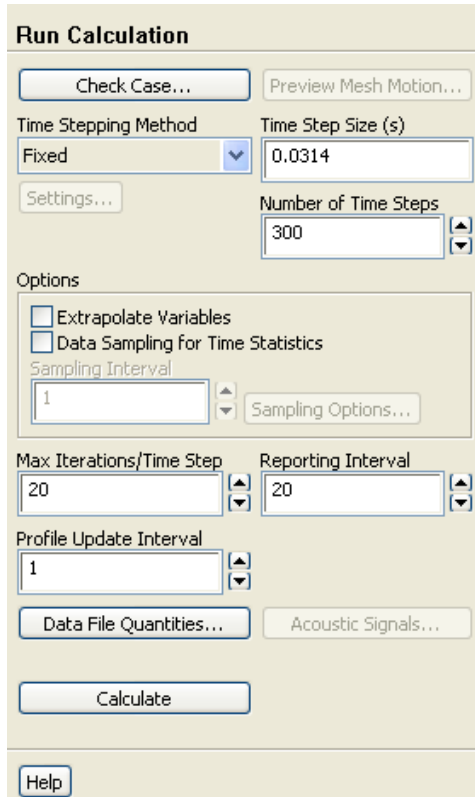
Figure 8.2.7: The Pressure Outlet Dialog Box

The time-stepping parameters are set in the Run Calculation task page (Figure 8.2.8).

✧ Run Calculation

Figure 8.2.8: The Run Calculation Task Page

In this example, a Time Step Size of 0.0314 s is used so that 20 time steps will complete a full period of oscillation in the outlet velocity. The Profile Update Interval is set to 1 so that the pressure will be updated every iteration. After 300 time steps (or 15 periods) are complete, you can examine the pressure and velocity magnitude across the pressure outlet.

To collect this information during the calculation, open the Surface Monitor dialog box (Figure 8.2.9) before beginning the calculation.

✧ **Monitors** ⟶ Create... (Surface Monitors)
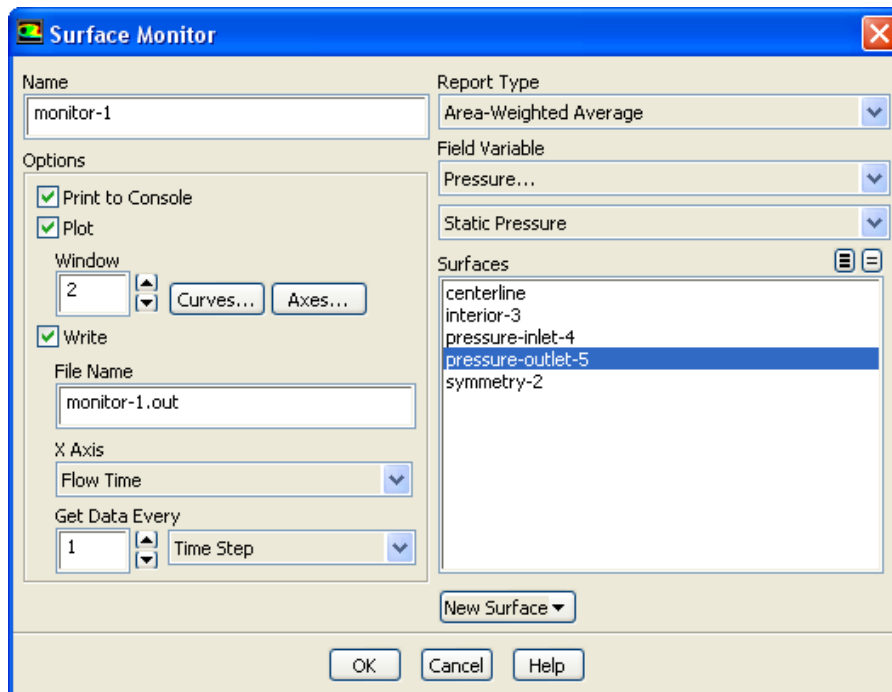
Figure 8.2.9: The **Surface Monitor** Dialog Box

The **Surface Monitor** dialog box will display the default settings. You can rename the surface monitor by entering `monitor-1` in the **Name** text box. Then set the parameters in the **Options** group box. Enable **Print to Console** to see the changing values of the selected quantity in the console. Enable **Plot** so that the selected quantity will be plotted as the calculation proceeds. Enable **Write** so that the information will be written to a file, which will be given the name you enter in the **File Name** text box (`monitor-1.out`). Select **Flow Time** from the **X Axis** drop-down list, and select **Time Step** in the drop-down list under **Get Data Every**.

Next, select **Area-Weighted Average** from the **Report Type** drop-down list. In the drop-down lists under **Field Variable**, select **Pressure...** and **Static Pressure**. Finally, select **pressure-outlet-5** in the **Surfaces** selection list and click **OK**

In a similar manner, you can set up a second monitor to capture the velocity magnitude fluctuations in the pressure outlet.

After the first time step has been completed, the monitors should appear in the chosen plot windows. Alternatively, you can read the files by opening the File XY Plot dialog box (Figure 8.2.10).

✛ Plots ⟶ ☰ File ⟶ Set Up...
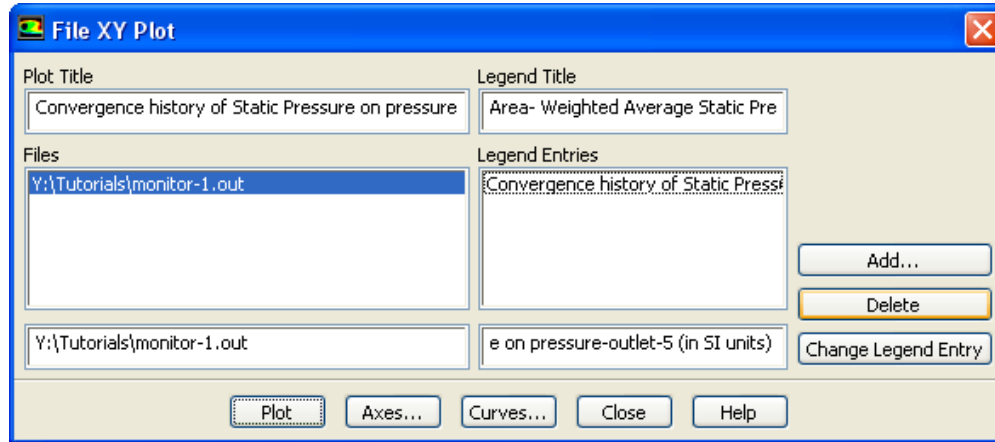
Figure 8.2.10: The File XY Plot Dialog Box

You can read an output file by clicking Add... and selecting it in the Select File dialog box that opens and clicking OK. Then click the Plot button in the File XY Plot dialog box to obtain plots like those shown in Figures 8.2.11 and 8.2.12.

Figure 8.2.11 nicely illustrates that the pressure oscillates around the equilibrium value, 101325 Pa, with an amplitude of 5 Pa, as expected.
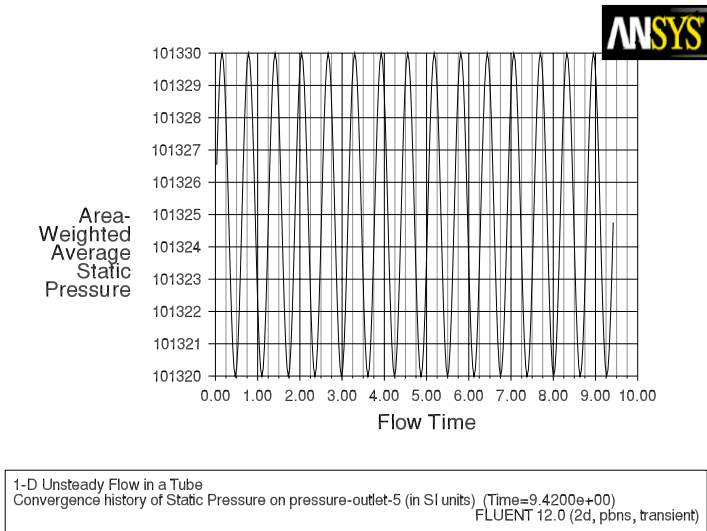
Figure 8.2.11: Average Static Pressure at the Pressure Outlet
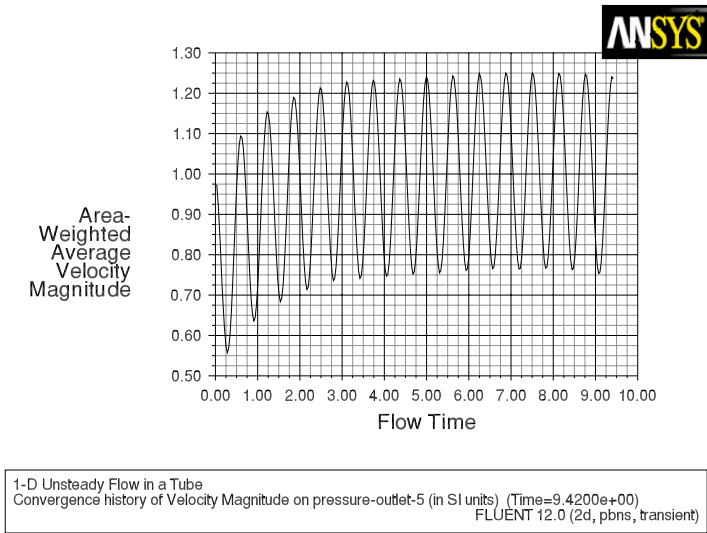
Figure 8.2.12: Average Velocity Magnitude at the Pressure Outlet

8.2.2 Source Terms

This section contains an application of a source term UDF. It is executed as an interpreted UDF in ANSYS FLUENT.

Adding a Momentum Source to a Duct Flow

When a source term is being modeled with a UDF, it is important to understand the context in which the function is called. When you add a source term, ANSYS FLUENT will call your function as it performs a global loop on cells. Your function should compute the source term and return it to the solver.

In this example, a momentum source will be added to a 2D Cartesian duct flow. The duct is 4 m long and 2 m wide, and will be modeled using a symmetry boundary through the middle. Liquid metal (with properties listed in Table 8.2.1) enters the duct at the left with a velocity of 1 mm/s at a temperature of 290 K. After the metal has traveled 0.5 m along the duct, it is exposed to a cooling wall, which is held at a constant temperature of 280 K. To simulate the freezing of the metal, a momentum source is applied to the metal as soon as its temperature falls below 288 K. The momentum source is proportional to the x component of the velocity, v_x, and has the opposite sign:

$$S_x = -Cv_x \qquad (8.2\text{-}1)$$

where C is a constant. As the liquid cools, its motion will be reduced to zero, simulating the formation of the solid. (In this simple example, the energy equation will not be customized to account for the latent heat of freezing. The velocity field will be used only as an indicator of the solidification region.)

The solver linearizes source terms in order to enhance the stability and convergence of a solution. To allow the solver to do this, you need to specify the dependent relationship between the source and solution variables in your UDF, in the form of derivatives. The source term, S_x, depends only on the solution variable, v_x. Its derivative with respect to v_x is

$$\frac{\partial S_x}{\partial v_x} = -C \qquad (8.2\text{-}2)$$

The following UDF specifies a source term and its derivative. The function, named `cell_x_source`, is defined on a cell using `DEFINE_SOURCE`. The constant C in Equation 8.2-1 is called `CON` in the function, and it is given a numerical value of 20 kg/m^3-s, which will result in the desired units of N/m^3 for the source. The temperature at the cell is returned by `C_T(cell,thread)`. The function checks to see if the temperature is below (or equal to) 288 K. If it is, the source is computed according to Equation 8.2-1 (`C_U` returns the value of the x velocity of the cell). If it is not, the source is set to 0. At

the end of the function, the appropriate value for the source is returned to the ANSYS FLUENT solver.

Table 8.2.1: Properties of the Liquid Metal

Property	Value
Density	8000 kg/m^3
Viscosity	5.5 $\times10^{-3}$ kg/m-s
Specific Heat	680 J/kg-K
Thermal Conductivity	30 W/m-K

```
/*******************************************************************
UDF that adds momentum source term and derivative to duct flow
*******************************************************************/

#include "udf.h"

#define CON 20.0

DEFINE_SOURCE(cell_x_source, cell, thread, dS, eqn)
{
  real source;

  if (C_T(cell,thread) <= 288.)
  {
    source = -CON*C_U(cell,thread);
    dS[eqn] = -CON;
  }
  else
  {
    source = dS[eqn] = 0.;
  }

  return source;
}
```

To make use of this UDF in ANSYS FLUENT, you will first need to interpret (or compile) the function, and then hook it to ANSYS FLUENT using the graphical user interface. Follow the procedure for interpreting source files using the Interpreted UDFs dialog box (Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box),

or compiling source files using the **Compiled UDFs** dialog box (Section 5.2: Compiling a UDF Using the GUI).

To include source terms in the calculation, you will first need to open the **Fluid** dialog box (Figure 8.2.13) by selecting the fluid zone in the **Cell Zone Conditions** task page and clicking **Edit**....

⟳ Cell Zone Conditions ⟶ ☰ fluid-8 ⟶ Edit...
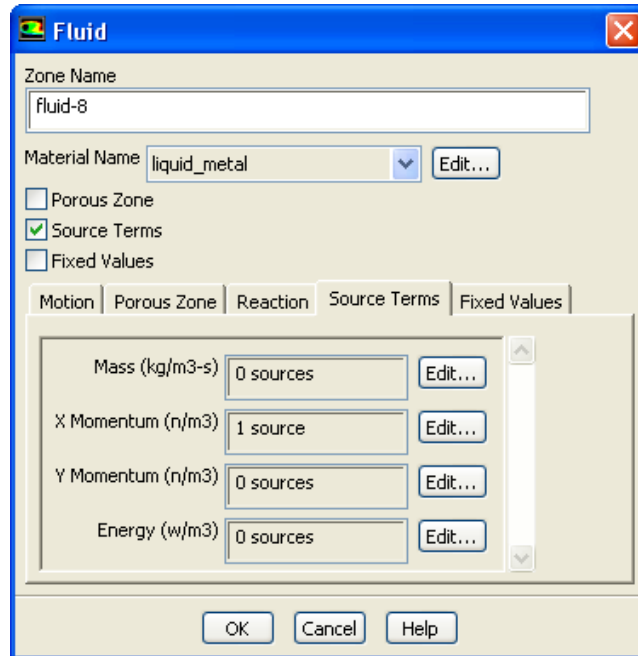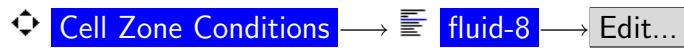
Figure 8.2.13: The **Fluid** Dialog Box

Enable the **Source Terms** option in the **Fluid** dialog box and click the **Source Terms** tab. This will display the momentum source term parameters in the scrollable window. Then, click the **Edit...** button next to the **X Momentum** source term to open the **X Momentum sources** dialog box (Figure 8.2.14).
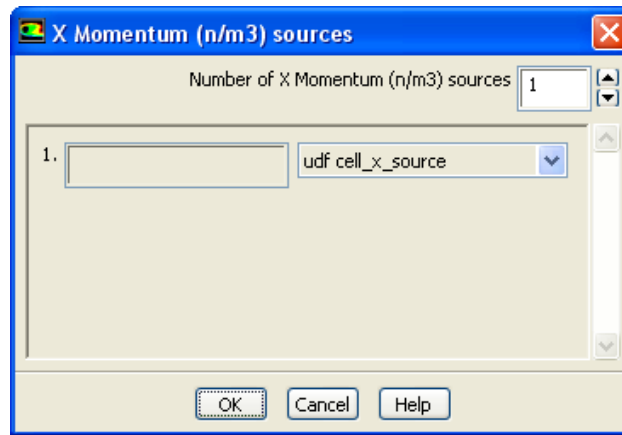
Figure 8.2.14: The X Momentum sources Dialog Box

Enter 1 for the Number of Momentum sources in the X Momentum sources dialog box and then select the function name for the UDF (udf cell_x_source) in the drop-down list that appears. (Note that the name that is displayed in the drop-down lists is your UDF name preceded by the word udf.) Click OK to accept the new cell zone condition and close the dialog box. The X Momentum parameter in the Fluid dialog box will now display 1 source. Click OK to fix the new momentum source term for the solution calculation and close the Fluid dialog box.

After the solution has converged, you can view contours of static temperature to see the cooling effects of the wall on the liquid metal as it moves through the duct (Figure 8.2.15).

Contours of velocity magnitude (Figure 8.2.16) show that the liquid in the cool region near the wall has indeed come to rest to simulate solidification taking place.
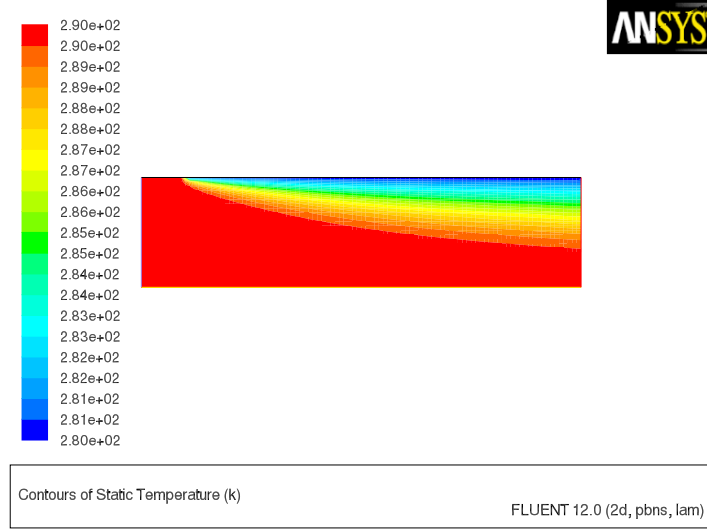
Figure 8.2.15: Temperature Contours Illustrating Liquid Metal Cooling
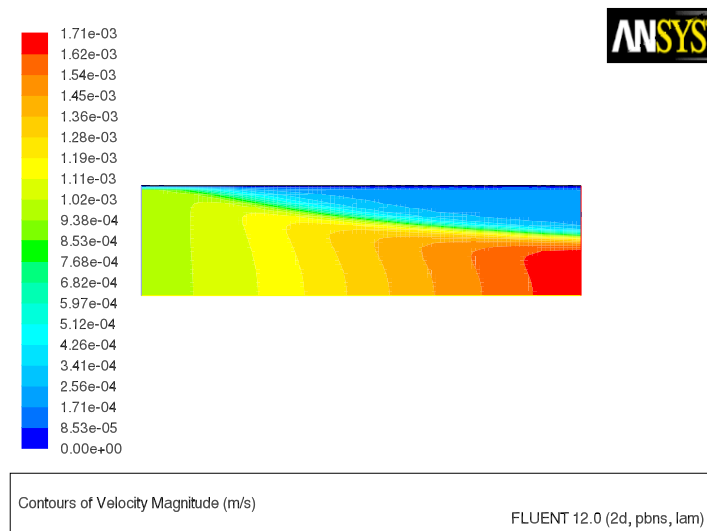
Figure 8.2.16: Velocity Magnitude Contours Suggesting Solidification

The solidification is further illustrated by line contours of stream function (Figure 8.2.17).
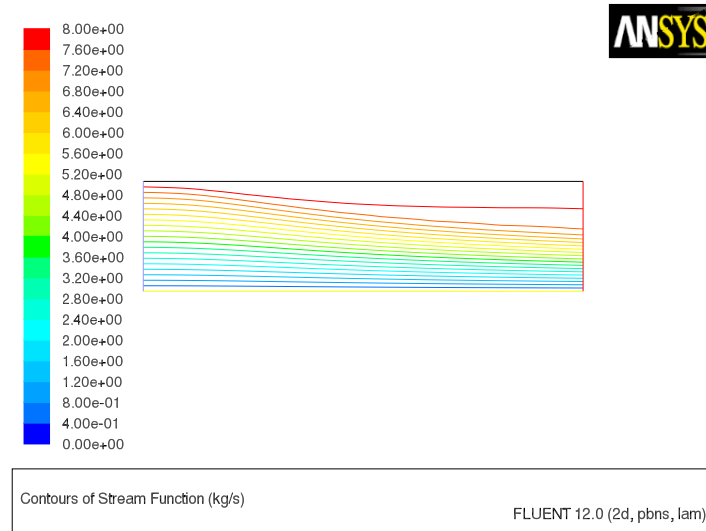
Figure 8.2.17: Stream Function Contours Suggesting Solidification

To more accurately predict the freezing of a liquid in this manner, an energy source term would be needed, as would a more accurate value for the constant appearing in Equation 8.2-1.

8.2.3   Physical Properties

This section contains an application of a physical property UDF. It is executed as an interpreted UDF in ANSYS FLUENT.

Solidification via a Temperature-Dependent Viscosity

UDFs for properties (as well as sources) are called from within a loop on cells. For this reason, functions that specify properties are only required to compute the property for a single cell, and return the value to the ANSYS FLUENT solver.

The UDF in this example generates a variable viscosity profile to simulate solidification, and is applied to the same problem that was presented in Section 8.2.2: Adding a Momentum Source to a Duct Flow. The viscosity in the warm ($T > 288$ K) fluid has a molecular value for the liquid (5.5×10^{-3} kg/m-s), while the viscosity for the cooler region ($T < 286$ K) has a much larger value (1.0 kg/m-s). In the intermediate temperature range (286 K $\leq T \leq 288$ K), the viscosity follows a linear profile (Equation 8.2-3) that extends between the two values given above:

$$\mu = 143.2135 - 0.49725T \qquad (8.2\text{-}3)$$

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The C source code for the UDF is shown below.

The function, named `cell_viscosity`, is defined on a cell using `DEFINE_PROPERTY`. Two `real` variables are introduced: `temp`, the value of `C_T(cell,thread)`, and `mu_lam`, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of `mu_lam` is computed. At the end of the function, the computed value for `mu_lam` is returned to the solver.

```
/************************************************************************
 UDF for specifying a temperature-dependent viscosity property
 ************************************************************************/

#include "udf.h"

DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
  real mu_lam;
  real temp = C_T(cell, thread);

  if (temp > 288.)
   mu_lam = 5.5e-3;
  else if (temp > 286.)
   mu_lam = 143.2135 - 0.49725 * temp;
  else
   mu_lam = 1.;

  return mu_lam;
}
```

This function can be executed as an interpreted or compiled UDF in ANSYS FLUENT. Follow the procedure for interpreting source files using the Interpreted UDFs dialog box (Section 4.2: Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box), or compiling source files using the Compiled UDFs dialog box (Section 5.2: Compiling a UDF Using the GUI)

To make use of the user-defined property in ANSYS FLUENT, you will need to open the Create/Edit Materials dialog box (Figure 8.2.18) by selecting the liquid metal material in the Materials task page and clicking the Create/Edit... button..

◇ Materials ⟶ 𝌆 liquid_metal ⟶ Create/Edit...
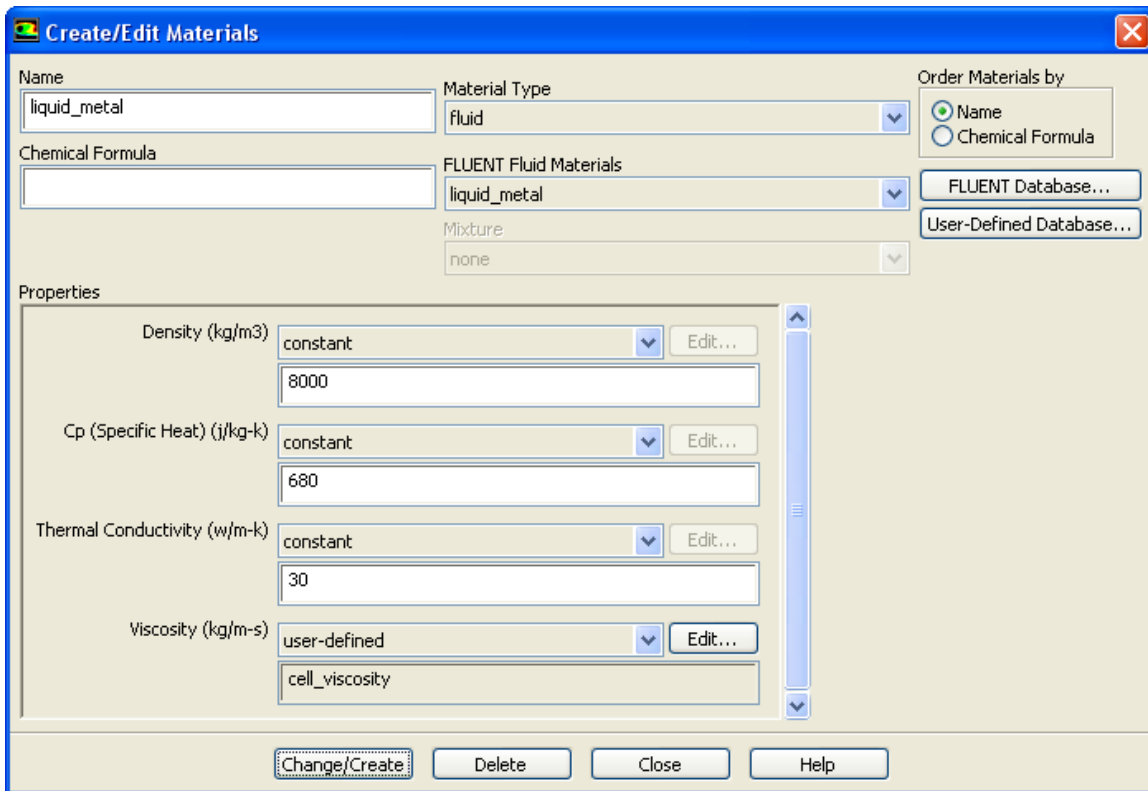
Figure 8.2.18: The Create/Edit Materials Dialog Box

In the Create/Edit Materials dialog box, select user-defined in the drop-down list for Viscosity. This will open the User-Defined Functions dialog box (Figure 8.2.19), from which you can select the appropriate function name. In this example, only one option is available, but in other examples, you may have several functions from which to choose. (Recall that if you need to compile more than one interpreted UDF, the functions can be concatenated in a single source file prior to compiling.)
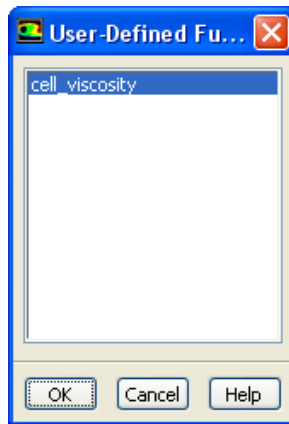
Figure 8.2.19: The User-Defined Functions Dialog Box

The results of this model are similar to those obtained in Section 8.2.2: Adding a Momentum Source to a Duct Flow. Figure 8.2.20 shows the viscosity field resulting from the application of the user-defined function. The viscosity varies rapidly over a narrow spatial band from a constant value of 0.0055 to 1.0 kg/m-s.

The velocity field (Figure 8.2.21) demonstrates that the liquid slows down in response to the increased viscosity, as expected. In this model, there is a large "mushy" region, in which the motion of the fluid gradually decreases. This is in contrast to the first model, in which a momentum source was applied and a more abrupt change in the fluid motion was observed.
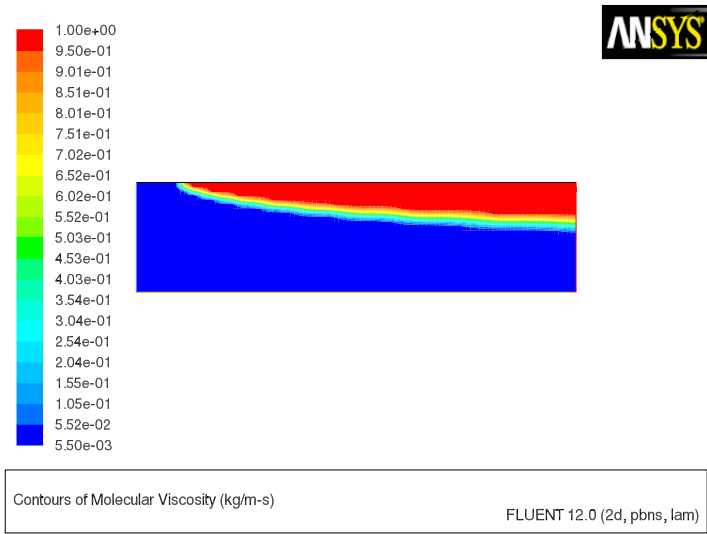
Figure 8.2.20: Laminar Viscosity Generated by a User-Defined Function
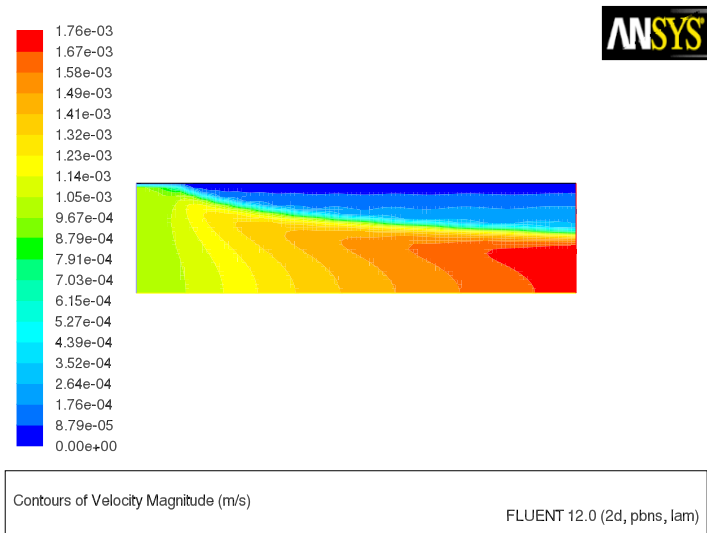
Figure 8.2.21: Contours of Velocity Magnitude Resulting from a User-Defined Viscosity
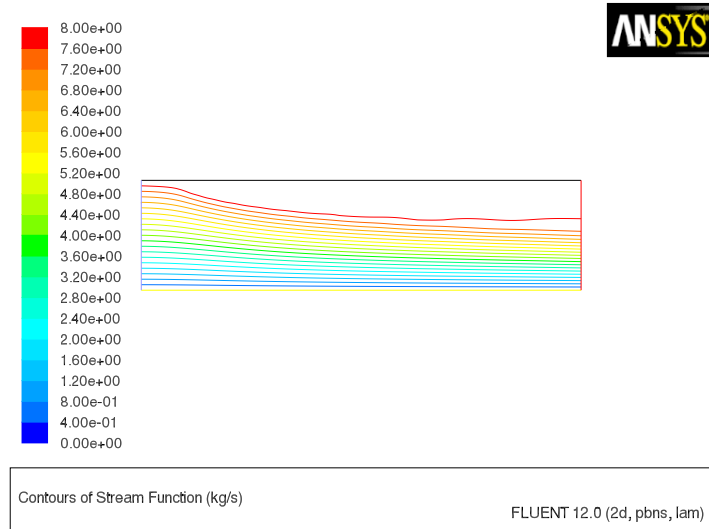
Figure 8.2.22: Stream Function Contours Suggesting Solidification

8.2.4 Reaction Rates

This section contains an example of a custom reaction rate UDF. It is executed as a compiled UDF in ANSYS FLUENT.

Volume Reaction Rate

A custom volume reaction rate for a simple system of two gaseous species is considered. The species are named `species-a` and `species-b`. The reaction rate is one that converts `species-a` into `species-b` at a rate given by the following expression:

$$R = \frac{K_1 X_a}{(1 + K_2 X_a)^2} \tag{8.2-4}$$

where X_a is the mass fraction of `species-a`, and K_1 and K_2 are constants.

The 2D (planar) domain consists of a 90-degree bend. The duct has a porous region covers the bottom and right-hand wall, and the reaction takes place in the porous region only. The species in the duct have identical properties. The density is 1.0 kg/m^3, and the viscosity is 1.7894×10^{-5} kg/m-s.

The outline of the domain is shown in Figure 8.2.23. The porous medium is the region below and to the right of the line that extends from the inlet on the left to the pressure outlet at the top of the domain.
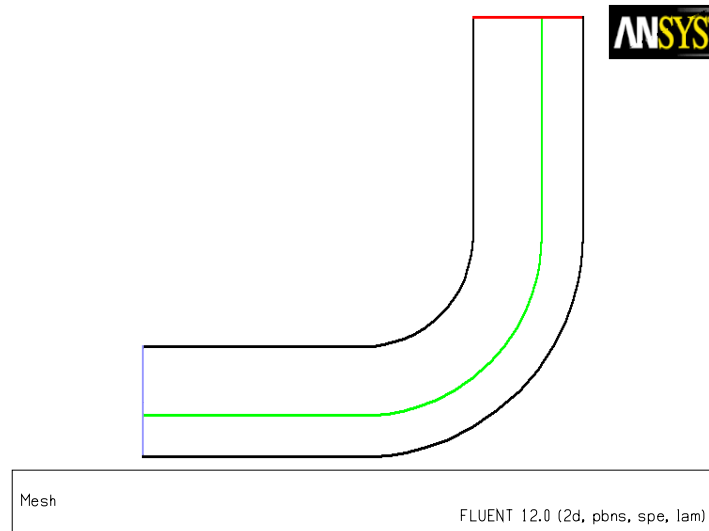
Figure 8.2.23: The Outline of the 2D Duct

Through the inlet on the left, gas that is purely `species-a` enters with an x velocity of 0.1 m/s. The gas enters both the open region on the top of the porous medium and the porous medium itself, where there is an inertial resistance of 5 m^{-1} in each of the two coordinate directions. The laminar flow field (Figure 8.2.24) shows that most of the gas is diverted from the porous region into the open region.

The flow pattern is further substantiated by the vector plot shown in Figure 8.2.25. The flow in the porous region is considerably slower than that in the open region.
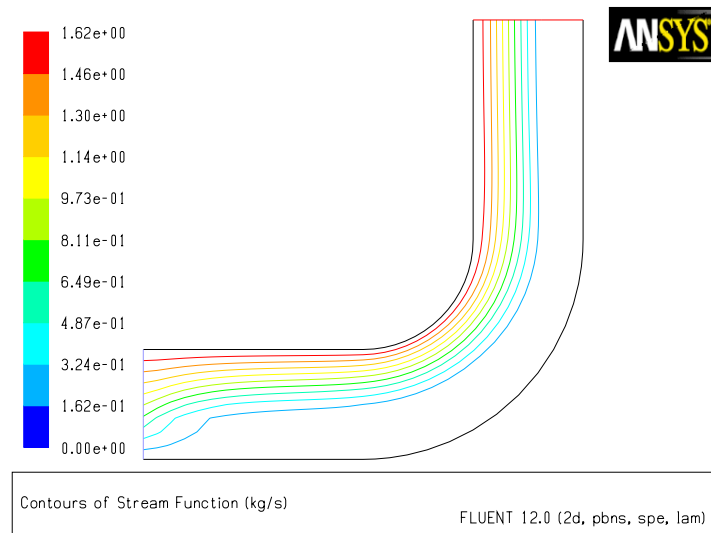
Figure 8.2.24: Streamlines for the 2D Duct with a Porous Region
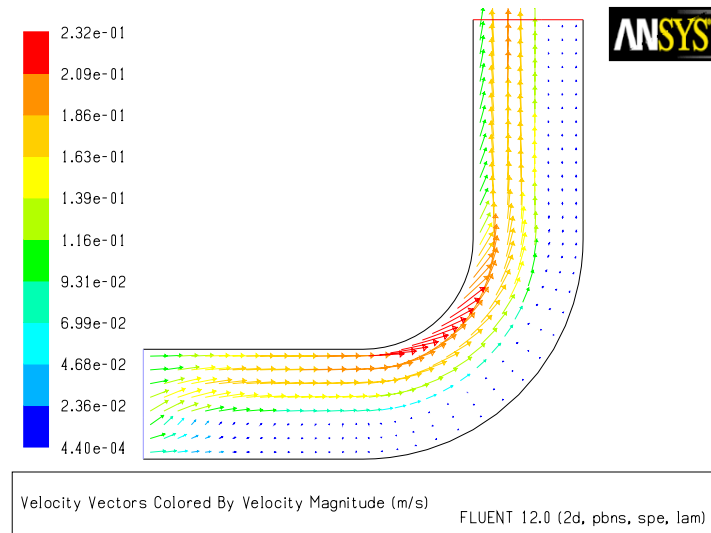
Figure 8.2.25: Velocity Vectors for the 2D Duct with a Porous Region

The source code (`rate.c`) that contains the UDF used to model the reaction taking place in the porous region is shown below. The function, named `vol_reac_rate`, is defined on a cell for a given species mass fraction using `DEFINE_VR_RATE`. The UDF performs a test to check for the porous region, and only applies the reaction rate equation to the porous region. The macro `FLUID_THREAD_P(t)` is used to determine if a cell thread is a fluid (rather than a solid) thread. The variable `THREAD_VAR(t).fluid.porous` is used to check if a fluid cell thread is a porous region.

```
/***********************************************************************
rate.c
Compiled UDF for specifying a reaction rate in a porous medium
***********************************************************************/

#include "udf.h"

#define K1 2.0e-2
#define K2 5.

DEFINE_VR_RATE(vol_reac_rate,c,t,r,mole_weight,species_mf,rate,rr_t)
{
  real s1 = species_mf[0];
  real mw1 = mole_weight[0];

  if (FLUID_THREAD_P(t) && THREAD_VAR(t).fluid.porous)
    *rate = K1*s1/pow((1.+K2*s1),2.0)/mw1;
  else
    *rate = 0.;

  *rr_t = *rate;
}
```

This UDF is executed as a compiled UDF in ANSYS FLUENT. Follow the procedure for compiling source files using the Compiled UDFs dialog box that is described in Section 5.2: Compiling a UDF Using the GUI.

After the function vol_reac_rate is compiled and loaded, you can hook the reaction rate UDF to ANSYS FLUENT by selecting the function's name in the Volume Reaction Rate Function drop-down list in the User-Defined Function Hooks dialog box (Figure 6.2.36).

Define ⟶ User-Defined ⟶ Function Hooks...
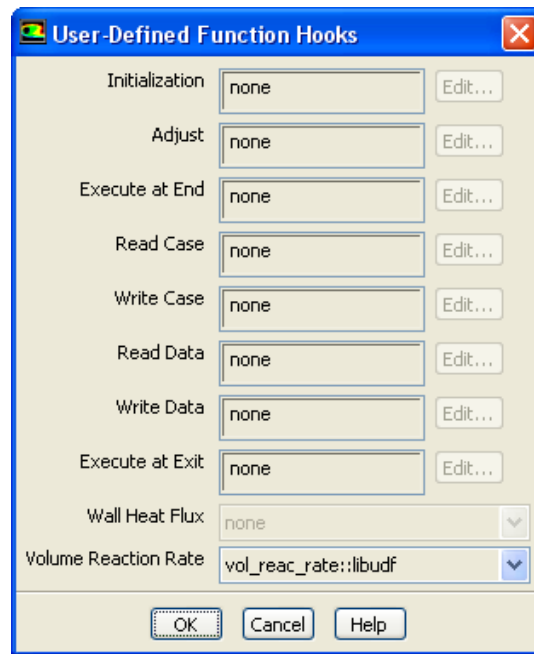
Figure 8.2.26: The User-Defined Functions Hooks Dialog Box

Initialize and run the calculation. The converged solution for the mass fraction of species-a is shown in Figure 8.2.27. The gas that moves through the porous region is gradually converted to species-b in the horizontal section of the duct. No reaction takes place in the fluid region, although some diffusion of species-b out of the porous region is suggested by the wide transition layer between the regions of 100% and 0% species-a.
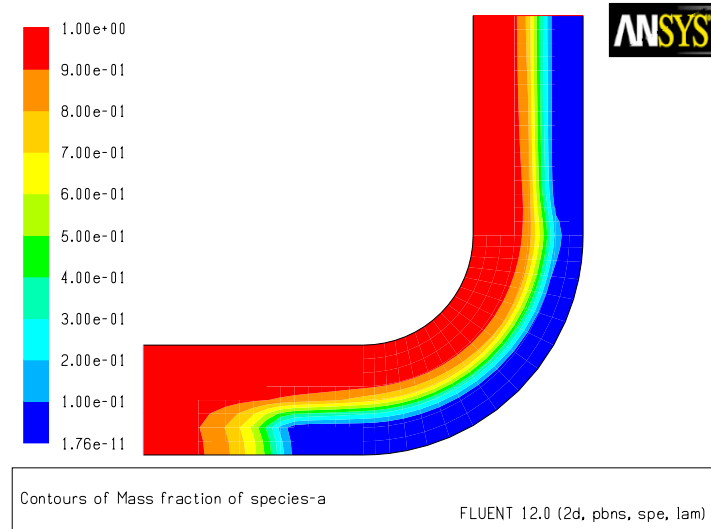
Figure 8.2.27: Mass Fraction for species-a Governed by a Reaction in a Porous Region

8.2.5 User-Defined Scalars

This section contains examples of UDFs that can be used to customize user-defined scalar (UDS) transport equations. See Section 2.7: User-Defined Scalar (UDS) Transport Equation DEFINE Macros in the UDF Manual for information on how you can define UDFs in ANSYS FLUENT. See Section 1.3: User-Defined Scalar (UDS) Transport Equations in the separate Theory Guide for UDS equation theory and details on how to set up scalar equations.

Postprocessing Using User-Defined Scalars

Below is an example of a compiled UDF that computes the gradient of temperature to the fourth power, and stores its magnitude in a user-defined scalar. The computed temperature gradient can, for example, be subsequently used to plot contours. Although the practical application of this UDF is questionable, its purpose here is to show the methodology of computing gradients of arbitrary quantities that can be used for postprocessing.

```
/**********************************************************************/
/* UDF for computing the magnitude of the gradient of T^4            */
/**********************************************************************/

#include "udf.h"
```

```
/* Define which user-defined scalars to use. */
enum
{
  T4,
  MAG_GRAD_T4,
  N_REQUIRED_UDS
};

DEFINE_ADJUST(adjust_fcn, domain)
{
  Thread *t;
  cell_t c;
  face_t f;


  /* Make sure there are enough user-defined scalars. */
  if (n_uds < N_REQUIRED_UDS)
    Internal_Error("not enough user-defined scalars allocated");

  /* Fill first UDS with temperature raised to fourth power. */
  thread_loop_c (t,domain)
    {
      if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)))
        {
          begin_c_loop (c,t)
            {
              real T = C_T(c,t);
              C_UDSI(c,t,T4) = pow(T,4.);
            }
          end_c_loop (c,t)
        }
    }


  thread_loop_f (t,domain)
    {
      if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)))
        {
          begin_f_loop (f,t)
            {
              real T = 0.;
              if (NULL != THREAD_STORAGE(t,SV_T))
```

```
                 T = F_T(f,t);
               else if (NULL != THREAD_STORAGE(t->t0,SV_T))
                 T = C_T(F_C0(f,t),t->t0);
               F_UDSI(f,t,T4) = pow(T,4.);
           }
         end_f_loop (f,t)
       }
   }

  /* Fill second UDS with magnitude of gradient. */
  thread_loop_c (t,domain)
    {
      if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)) &&
          NULL != T_STORAGE_R_NV(t,SV_UDSI_G(T4)))
      {
         begin_c_loop (c,t)
           {
              C_UDSI(c,t,MAG_GRAD_T4) = NV_MAG(C_UDSI_G(c,t,T4));
           }
         end_c_loop (c,t)
      }
    }

  thread_loop_f (t,domain)
    {
      if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)) &&
          NULL != T_STORAGE_R_NV(t->t0,SV_UDSI_G(T4)))
      {
         begin_f_loop (f,t)
          {
           F_UDSI(f,t,MAG_GRAD_T4)=C_UDSI(F_C0(f,t),t->t0,MAG_GRAD_T4);
          }
         end_f_loop (f,t)
      }
    }
}
```

The conditional statement `if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)))` is used to check if the storage for the user-defined scalar with index `T4` has been allocated, while `NULL != T_STORAGE_R_NV(t,SV_UDSI_G(T4))` checks whether the storage of the gradient of the user-defined scalar with index `T4` has been allocated.

In addition to compiling this UDF, as described in Chapter 5: Compiling UDFs, you will need to enable the solution of a user-defined scalar transport equation in ANSYS FLUENT.

Define $\longrightarrow$ User-Defined $\longrightarrow$ Scalars...

See Section 1.3: User-Defined Scalar (UDS) Transport Equations in the separate Theory Guide for UDS equation theory and details on how to setup scalar equations.

Implementing ANSYS FLUENT's P-1 Radiation Model Using User-Defined Scalars

This section provides an example that demonstrates how the P1 radiation model can be implemented as a UDF, utilizing a user-defined scalar transport equation. In the P1 model, the variation of the incident radiation, G, in the domain can be described by an equation that consists of a diffusion and source term.

The transport equation for incident radiation, G, is given by Equation 8.2-5. The diffusion coefficient, Γ, is given by Equation 8.2-6 and the source term is given by Equation 8.2-7. See Section 5.3.3: P-1 Radiation Model Theory in the separate Theory Guide for more details.

$$\nabla \cdot (\Gamma \nabla G) + S^G = 0 \qquad (8.2\text{-}5)$$

$$\Gamma = \frac{1}{3a + (3 - C)\,\sigma_s} \qquad (8.2\text{-}6)$$

$$S^G = a\left(4\sigma T^4 - G\right) \qquad (8.2\text{-}7)$$

The boundary condition for G at the walls is equal to the negative of the radiative wall heat flux, $q_{r,w}$ (Equation 8.2-8), where $\vec{n}$ is the outward normal vector (see Section 5.3.3: P-1 Radiation Model Theory in the separate Theory Guide for more details). The radiative wall heat flux can be given by Equation 8.2-9.

$$q_r \cdot \vec{n} = -\Gamma \nabla G \cdot \vec{n} \qquad (8.2\text{-}8)$$

$$q_{r,w} = -\frac{\epsilon_w}{2\left(2 - \epsilon_w\right)}\left(4\sigma T_w^4 - G_w\right) \qquad (8.2\text{-}9)$$

This form of the boundary condition is unfortunately specified in terms of the incident radiation at the wall, G_w. This mixed boundary condition can be avoided by solving first for G_w using Equations 8.2-8 and 8.2-9, resulting in Equation 8.2-10. Then, this

expression for G_w is substituted back into Equation 8.2-9 to give the radiative wall heat flux $q_{r,w}$ as Equation 8.2-11.

$$G_w = \frac{4\sigma T_w^4 E_w + \frac{\alpha_0 \Gamma_0}{A}\left[G_0 - \beta_0(G)\right]}{E_w + \frac{\alpha_0 \Gamma_0}{A}} \qquad (8.2\text{-}10)$$

$$q_r = -\frac{\alpha_0 \Gamma_0 E_w}{A\left(E_w + \frac{\alpha_0 \Gamma_0}{A}\right)}\left[4\pi I_b(T_{iw}) - G_0 + \beta_0(G)\right] \qquad (8.2\text{-}11)$$

The additional β_0 and G_0 terms that appear in Equations 8.2-10 and 8.2-11 are a result of the evaluation of the gradient of incident radiation in Equation 8.2-8.

In ANSYS FLUENT, the component of a gradient of a scalar directed normal to a cell boundary (face), $\nabla G \cdot \mathbf{n}$, is estimated as the sum of primary and secondary components. The primary component represents the gradient in the direction defined by the cell centroids, and the secondary component is in the direction along the face separating the two cells. From this information, the face normal component can be determined. The secondary component of the gradient can be found using the ANSYS FLUENT macro BOUNDARY_SECONDARY_GRADIENT_SOURCE (which is described in Section 3.2.5: Boundary Secondary Gradient Source (BOUNDARY_SECONDARY_GRADIENT_SOURCE)). The use of this macro first requires that cell geometry information be defined, which can be readily obtained by the use of a second macro, BOUNDARY_FACE_GEOMETRY (see Section 3.2.5: Boundary Face Geometry (BOUNDARY_FACE_GEOMETRY)). You will see these macros called in the UDF that defines the wall boundary condition for G.

To complete the implementation of the P1 model, the radiation energy equation must be coupled with the thermal energy equation. This is accomplished by modifying the source term and wall boundary condition of the energy equation. Consider first how the energy equation source term must be modified. The gradient of the incident radiation is proportional to the radiative heat flux. A local increase (or decrease) in the radiative heat flux is attributable to a local decrease (or increase) in thermal energy via the absorption and emission mechanisms. The gradient of the radiative heat flux is therefore a (negative) source of thermal energy. The source term for the incident radiation Equation 8.2-7 is equal to the gradient of the radiative heat flux and hence its negative specifies the source term needed to modify the energy equation (see Section 5.3.3: P-1 Radiation Model Theory in the separate Theory Guide for more details).

Now consider how the energy boundary condition at the wall must be modified. Locally, the only mode of energy transfer from the wall to the fluid that is accounted for by default is conduction. With the inclusion of radiation effects, radiative heat transfer to and from the wall must also be accounted for. (This is done automatically if you use ANSYS FLUENT's built-in P1 model.) The DEFINE_HEAT_FLUX macro allows the wall boundary condition to be modified to accommodate this second mode of heat transfer by specifying the coefficients of the qir equation discussed in Section 2.3.9: DEFINE_HEAT_FLUX. The net

radiative heat flux to the wall has already been given as Equation 8.2-9. Comparing this equation with that for qir in Section 2.3.9: DEFINE_HEAT_FLUX will result in the proper coefficients for $cir[]$.

In this example, the implementation of the P1 model can be accomplished through six separate UDFs. They are all included in a single source file, which can be executed as a compiled UDF. The single user-defined scalar transport equation for incident radiation, G, uses a DEFINE_DIFFUSIVITY UDF to define Γ of Equation 8.2-6, and a UDF to define the source term of Equation 8.2-7. The boundary condition for G at the walls is handled by assigning, in DEFINE_PROFILE, the negative of Equation 8.2-11 as the specified flux. A DEFINE_ADJUST UDF is used to instruct ANSYS FLUENT to check that the proper number of user-defined scalars has been defined (in the solver). Lastly, the energy equation must be assigned a source term equal to the negative of that used in the incident radiation equation and the DEFINE_HEAT_FLUX UDF is used to alter the boundary conditions at the walls for the energy equation.

In the solver, at least one user-defined scalar (UDS) equation must be enabled. The scalar diffusivity is assigned in the Create/Edit Materials dialog box for the scalar equation. The scalar source and energy source terms are assigned in the boundary condition dialog box for the fluid zones. The boundary condition for the scalar equation at the walls is assigned in the boundary condition dialog box for the wall zones. The DEFINE_ADJUST and DEFINE_HEAT_FLUX functions are assigned in the User-Defined Function Hooks dialog box.

Note that the residual monitor for the UDS equation should be reduced from $1e - 3$ to $1e - 6$ before running the solution. If the solution diverges, then it may be due to the large source terms. In this case, the under-relaxation factor should be reduced to 0.99 and the solution re-run.

```
/***********************************************************/
/* Implementation of the P1 model using user-defined scalars  */
/***********************************************************/

#include "udf.h"
#include "sg.h"

/* Define which user-defined scalars to use. */
enum
{
  P1,
  N_REQUIRED_UDS
};


static real abs_coeff = 0.2;     /* absorption coefficient */
```

```
static real scat_coeff = 0.0;    /* scattering coefficient */
static real las_coeff = 0.0;     /* linear-anisotropic    */
                                 /* scattering coefficient */
static real epsilon_w = 1.0;     /* wall emissivity */


DEFINE_ADJUST(p1_adjust, domain)
{
  /* Make sure there are enough user defined-scalars. */
  if (n_uds < N_REQUIRED_UDS)
    Internal_Error("not enough user-defined scalars allocated");
}


DEFINE_SOURCE(energy_source, c, t, dS, eqn)
{
  dS[eqn] = -16.*abs_coeff*SIGMA_SBC*pow(C_T(c,t),3.);
  return -abs_coeff*(4.*SIGMA_SBC*pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
}


DEFINE_SOURCE(p1_source, c, t, dS, eqn)
{
  dS[eqn] = 16.*abs_coeff*SIGMA_SBC*pow(C_T(c,t),3.);
  return abs_coeff*(4.*SIGMA_SBC*pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
}


DEFINE_DIFFUSIVITY(p1_diffusivity, c, t, i)
{
  return 1./(3.*abs_coeff + (3. - las_coeff)*scat_coeff);
}


DEFINE_PROFILE(p1_bc, thread, position)

{
  face_t f;
  real A[ND_ND],At;
  real dG[ND_ND],dr0[ND_ND],es[ND_ND],ds,A_by_es;
  real aterm,alpha0,beta0,gamma0,Gsource,Ibw;
  real Ew = epsilon_w/(2.*(2. - epsilon_w));
  Thread *t0=thread->t0;
```

```
  /* Do nothing if areas aren't computed yet or not next to fluid. */
  if (!Data_Valid_P() || !FLUID_THREAD_P(t0)) return;

  begin_f_loop (f,thread)
    {
      cell_t c0 = F_C0(f,thread);

      BOUNDARY_FACE_GEOMETRY(f,thread,A,ds,es,A_by_es,dr0);
      At = NV_MAG(A);

      if (NULLP(T_STORAGE_R_NV(t0,SV_UDSI_G(P1))))
        Gsource = 0.;    /* if gradient not stored yet */
      else
        BOUNDARY_SECONDARY_GRADIENT_SOURCE(Gsource,SV_UDSI_G(P1),
                                           dG,es,A_by_es,1.);

      gamma0 = C_UDSI_DIFF(c0,t0,P1);
      alpha0 = A_by_es/ds;
      beta0  = Gsource/alpha0;
      aterm  = alpha0*gamma0/At;

      Ibw = SIGMA_SBC*pow(WALL_TEMP_OUTER(f,thread),4.)/M_PI;

      /* Specify the radiative heat flux. */
      F_PROFILE(f,thread,position) =
        aterm*Ew/(Ew + aterm)*(4.*M_PI*Ibw - C_UDSI(c0,t0,P1) + beta0);
    }
  end_f_loop (f,thread)
}


DEFINE_HEAT_FLUX(heat_flux, f, t, c0, t0, cid, cir)
{

    real Ew = epsilon_w/(2.*(2. - epsilon_w));

    cir[0] = Ew * F_UDSI(f,t,P1);
    cir[3] = 4.0 * Ew * SIGMA_SBC;

}
```

8.2.6 User-Defined Real Gas Models

This section contains examples of UDFs that can be used to customize user-defined real gas models. See Section 8.16.3: The User-Defined Real Gas Model in the separate User's Guide for the UDRGM overview, limitations, and details on how to set up, build and load a library of user-defined real gas functions.

UDRGM Example: Redlich-Kwong Equation of State

This section describes another example of a user-defined real gas model. You can use this example as the basis for your own UDRGM code. In this example, the Redlich-Kwong equation of state is used in the UDRGM.

This section summarizes the equations used in developing the UDRGM for the Redlich-Kwong equation of state. The model is based on a modified form of the Redlich-Kwong equation of state described in [1]. The equations used in this UDRGM will be listed in the sections below.

The following nomenclature applies to this section:

$a(T)$ = Redlich-Kwong temperature function
c = speed of sound
C_p = specific heat
H = enthalpy
n = exponent in function $a(T)$
p = pressure
R = universal gas constant/molecular weight
T = temperature
S = entropy
V = specific volume
ρ = density

The superscript 0 designates a reference state, and the subscript c designates a critical point property.

Specific Volume and Density

The Redlich-Kwong equation of state can be written in the following form:

$$p = \frac{RT}{(V - \tilde{b})} - \frac{a(T)}{V(V + b_0)} \tag{8.2-12}$$

where

$$V = \frac{1}{\rho}, \quad a(T) = a_0 \left(\frac{T_c}{T}\right)^n, \quad a_0 = 0.42747 \frac{R^2 T_c^2}{p_c},$$

$$b_0 = 0.08664 \frac{RT_c}{p_c}, \quad c_0 = \frac{RT_c}{p_c + \frac{a_0}{V_c(V_c + b_0)}} + b_0 - V_c, \quad \tilde{b} = b_0 - c_0$$

Since the real gas model in ANSYS FLUENT requires a function for density as a function of pressure and temperature, Equation 8.2-12 must be solved for the specific volume (from which the density can be easily obtained). For convenience, Equation 8.2-12 can be written as a cubic equation for specific volume as follows:

$$V^3 + a_1 V^2 + a_2 V + a_3 = 0 \tag{8.2-13}$$

where

$$a_1 = c_0 - \frac{RT}{p}, \quad a_2 = -\left(\tilde{b} b_0 + \frac{RT b_0}{p} - \frac{a(T)}{p}\right), \quad a_3 = -\frac{a(T)\,\tilde{b}}{p}$$

Equation 8.2-13 is solved using a standard algorithm for cubic equations (see [10] for details). In the UDRGM code, the cubic solution is coded to minimize the number of floating point operations. This is critical for optimal performance, since this function gets called numerous times during an iteration cycle.

It should be noted that the value of the exponent, n, in the function $a(T)$ will depend on the substance. A table of values can be found in [1] for some common substances. Alternatively, [1] states that values of n are well correlated by the empirical equation

$$n = 0.4986 + 1.1735\omega + 0.475\omega^2 \tag{8.2-14}$$

where ω is the acentric factor, defined as

$$\omega = -\log\left(\frac{p_v(T)}{p_c}\right) - 1.0 \tag{8.2-15}$$

In the above equation, $p_v(T)$ is the saturated vapor pressure evaluated at temperature $T = 0.7 T_c$.

Derivatives of Specific Volume and Density

The derivatives of specific volume with respect to temperature and pressure can be easily determined from Equation 8.2-12 using implicit differentiation. The results are presented below:

$$\left(\frac{\partial V}{\partial p}\right)_T = -\frac{(a_1)'_p V^2 + (a_2)'_p V + (a_3)'_p}{3V^2 + 2a_1 V + a_2} \tag{8.2-16}$$

$$\left(\frac{\partial V}{\partial T}\right)_p = -\frac{(a_1)'_T V^2 + (a_2)'_T V + (a_3)'_T}{3V^2 + 2a_1 V + a_2} \tag{8.2-17}$$

where

$$(a_1)'_p = \frac{RT}{p^2}, \quad (a_2)'_p = \frac{RTb_0 - a(T)}{p^2}, \quad (a_3)'_p = \frac{a(T)\,\tilde{b}}{p^2}$$

$$(a_1)'_T = -\frac{R}{p}, \quad (a_2)'_T = \frac{-Rb_0 + \frac{da(T)}{dT}}{p}, \quad (a_3)'_T = -\frac{da(T)}{dT}\frac{\tilde{b}}{p}, \quad \frac{da(T)}{dT} = -n\frac{a(T)}{T}$$

The derivatives of density can be obtained from the above using the relations

$$\left(\frac{\partial \rho}{\partial p}\right)_T = -\rho^2 \left(\frac{\partial V}{\partial p}\right)_T \tag{8.2-18}$$

$$\left(\frac{\partial \rho}{\partial T}\right)_p = -\rho^2 \left(\frac{\partial V}{\partial T}\right)_p \tag{8.2-19}$$

Specific Heat and Enthalpy

Following [1], enthalpy for a real gas can be written

$$H = H^0(T) + pV - RT - \frac{a(T)}{b_0}(1+n)\ln\left(\frac{V + b_0}{V}\right) \tag{8.2-20}$$

where $H^0(T)$ is the enthalpy function for a thermally perfect gas (i.e., enthalpy is a function of temperature alone). In the present case, we employ a fourth-order polynomial for the specific heat for a thermally perfect gas [8]

$$C_p^0(T) = C_1 + C_2 T + C_3 T^2 + C_4 T^3 + C_5 T^4 \tag{8.2-21}$$

and obtain the enthalpy from the basic relation

$$H^0(T) = \int_{T_0}^{T} C_p^0(T) dT \tag{8.2-22}$$

The result is

$$H^0(T) = C_1 T + \frac{1}{2}C_2 T^2 + \frac{1}{3}C_3 T^3 + \frac{1}{4}C_4 T^4 + \frac{1}{5}C_5 T^5 - H^0(T^0) \tag{8.2-23}$$

Note that $H^0(T^0)$ is the enthalpy at a reference state (p^0, T^0), which can be chosen arbitrarily.

The specific heat for the real gas can be obtained by differentiating Equation 8.2-20 with respect to temperature (at constant pressure):

$$C_p = \left(\frac{\partial H}{\partial T}\right)_p \tag{8.2-24}$$

The result is

$$C_p = C_p^0(T) + p\left(\frac{\partial V}{\partial T}\right)_p - R - \frac{da(T)}{dT}\frac{(1+n)}{b_0}\ln\left(\frac{V+b_0}{V}\right) + a(T)(1+n)\frac{\left(\frac{\partial V}{\partial T}\right)_p}{V(V+b_0)} \tag{8.2-25}$$

Finally, the derivative of enthalpy with respect to pressure (at constant temperature) can be obtained using the following thermodynamic relation [8]:

$$\left(\frac{\partial H}{\partial p}\right)_T = V - T\left(\frac{\partial V}{\partial T}\right)_p \tag{8.2-26}$$

Entropy

Following [1], the entropy can be expressed in the form

$$S = S^0(T, p^0) + R\ln\left(\frac{V-\tilde{b}}{V^0}\right) + \frac{\left(\frac{da(T)}{dT}\right)}{b_0}\ln\left(\frac{V+b_0}{V}\right) \tag{8.2-27}$$

where the superscript 0 again refers to a reference state where the ideal gas law is applicable. For an ideal gas at a fixed reference pressure, p^0 , the entropy is given by

$$S^0(T, p^0) = S(T^0, p^0) + \int_{T^0}^{T} \frac{C_p^0(T)}{T} dT \tag{8.2-28}$$

Note that the pressure term is zero since the entropy is evaluated at the reference pressure. Using the polynomial expression for specific heat, Equation 8.2-21, Equation 8.2-28 becomes

$$S^0(T, p^0) = S(T^0, p^0) + C_1 \ln(T) + C_2 T + \frac{1}{2}C_3 T^2 + \frac{1}{3}C_4 T^3 + \frac{1}{4}C_5 T^4 - f(T^0) \tag{8.2-29}$$

where $f(T^0)$ is a constant, which can be absorbed into the reference entropy $S(T^0, p^0)$.

Speed of Sound

The speed of sound for a real gas can be determined from the thermodynamic relation

$$c^2 = \left(\frac{\partial p}{\partial \rho}\right)_S = -\left(\frac{C_p}{C_V}\right) \frac{V^2}{\left(\frac{\partial V}{\partial p}\right)_T} \tag{8.2-30}$$

Noting that,

$$C_p - C_V = -T \left(\left.\frac{\partial V}{\partial T}\right|_p\right)^2 \left.\frac{\partial p}{\partial v}\right|_T \tag{8.2-31}$$

we can write the speed of sound as

$$c = V \sqrt{-\left(\frac{C_p}{C_p - \Delta C}\right) \frac{1}{\left(\frac{\partial V}{\partial p}\right)_T}} \tag{8.2-32}$$

Viscosity and Thermal Conductivity

The dynamic viscosity of a gas or vapor can be estimated using the following formula from [2]:

$$\mu(T) = 6.3 \times 10^7 \frac{M_w^{0.5} p_c^{0.6666}}{T_c^{0.1666}} \left(\frac{T_r^{1.5}}{T_r + 0.8}\right) \tag{8.2-33}$$

Here, T_r is the reduced temperature

$$T_r = \frac{T}{T_c} \tag{8.2-34}$$

and M_w is the molecular weight of the gas. This formula neglects the effect of pressure on viscosity, which usually becomes significant only at very high pressures.

Knowing the viscosity, the thermal conductivity can be estimated using the Eucken formula [4]:

$$k = \mu \left(C_p + \frac{5}{4} R \right) \tag{8.2-35}$$

It should be noted that both Equation 8.2-33 and 8.2-35 are simple relations, and therefore may not provide satisfactory values of viscosity and thermal conductivity for certain applications. You are encouraged to modify these functions in the UDRGM source code if alternate formulae are available for a given gas.

Using the Redlich-Kwong Real Gas UDRGM

Using the Redlich-Kwong Real Gas UDRGM simply requires the modification of the top block of `#define` macros to provide the appropriate parameters for a given substance. An example listing for CO_2 is given below. The parameters required are:

`MWT`	=	Molecular weight of the substance
`PCRIT`	=	Critical pressure (Pa)
`TCRIT`	=	Critical temperature (K)
`ZCRIT`	=	Critical compressibility factor
`VCRIT`	=	Critical specific volume ($\mathrm{m^3/kg}$)
`NRK`	=	Exponent of $a(T)$ function
`CC1, CC2, CC3, CC4, CC5`	=	Coefficients of $C_p(T)$ polynomial curve fit
`P_REF`	=	Reference pressure (Pa)
`T_REF`	=	Reference temperature (K)

The coefficients for the ideal gas specific heat polynomial were obtained from [8] (coefficients for other substances are also provided in [8]). After the source listing is modified, the UDRGM C code can be recompiled and loaded into ANSYS FLUENT in the manner described earlier.

```
/* The variables below need to be set for a particular gas */

/* CO2 */

/* REAL GAS EQUATION OF STATE MODEL - BASIC VARIABLES */
/* ALL VARIABLES ARE IN SI UNITS! */

#define RGASU UNIVERSAL_GAS_CONSTANT
#define PI    3.141592654

#define MWT 44.01
#define PCRIT 7.3834e6
#define TCRIT 304.21
#define ZCRIT 0.2769
#define VCRIT 2.15517e-3
#define NRK 0.77

/* IDEAL GAS SPECIFIC HEAT CURVE FIT */

#define CC1 453.577
#define CC2 1.65014
#define CC3 -1.24814e-3
#define CC4 3.78201e-7
#define CC5 0.00

/* REFERENCE STATE */

#define P_REF  101325
#define T_REF  288.15
```

Redlich-Kwong Real Gas UDRGM Code Listing

```
/*****************************************************************/
/*                                                               */
/* User-Defined Function: Redlich-Kwong Equation of State        */
/*                        for Real Gas Modeling                  */
/*                                                               */
/*  Author: Frank Kelecy                                         */
/*    Date: May 2003                                             */
/* Version: 1.02                                                 */
/*                                                               */
/* This implementation is completely general.                   */
/* Parameters set for CO2.                                       */
/*                                                               */
/*****************************************************************/

#include "udf.h"
#include "stdio.h"
#include "ctype.h"
#include "stdarg.h"


/* The variables below need to be set for a particular gas */

/* CO2 */

/* REAL GAS EQUATION OF STATE MODEL - BASIC VARIABLES */
/* ALL VARIABLES ARE IN SI UNITS! */

#define RGASU UNIVERSAL_GAS_CONSTANT
#define PI    3.141592654

#define MWT 44.01
#define PCRIT 7.3834e6
#define TCRIT 304.21
#define ZCRIT 0.2769
#define VCRIT 2.15517e-3
#define NRK 0.77
```

```
/* IDEAL GAS SPECIFIC HEAT CURVE FIT */

#define CC1 453.577
#define CC2 1.65014
#define CC3 -1.24814e-3
#define CC4 3.78201e-7
#define CC5 0.00

/* REFERENCE STATE */

#define P_REF   101325
#define T_REF   288.15

/* OPTIONAL REFERENCE (OFFSET) VALUES FOR ENTHALPY AND ENTROPY */

#define H_REF  0.0
#define S_REF  0.0

static int  (*usersMessage)(char *,...);
static void (*usersError)(char *,...);

/* Static variables associated with Redlich-Kwong Model */

static double rgas, a0, b0, c0, bb, cp_int_ref;


DEFINE_ON_DEMAND(I_do_nothing)
{
  /* this is a dummy function to allow us */
  /* to use the compiled UDFs utility     */
}



/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_error                                    */
/*----------------------------------------------------------*/

void RKEOS_error(int err, char *f, char *msg)
{
  if (err)
     usersError("RKEOS_error (%d) from function: %s\n%s\n",err,f,msg);
}
```

```
/*---------------------------------------------------------*/
/* FUNCTION: RKEOS_Setup                                   */
/*---------------------------------------------------------*/

void RKEOS_Setup(Domain *domain, cxboolean vapor_phase, char *filename, int
                 (*messagefunc)(char *format, ...),
                 void (*errorfunc)(char *format, ...))
{

   rgas = RGASU/MWT;
   a0 = 0.42747*rgas*rgas*TCRIT*TCRIT/PCRIT;
   b0 = 0.08664*rgas*TCRIT/PCRIT;
   c0 = rgas*TCRIT/(PCRIT+a0/(VCRIT*(VCRIT+b0)))+b0-VCRIT;
   bb = b0-c0;
   cp_int_ref = CC1*log(T_REF)+T_REF*(CC2+
               T_REF*(0.5*CC3+T_REF*(0.333333*CC4+0.25*CC5*T_REF)));

   usersMessage = messagefunc;
   usersError = errorfunc;
   usersMessage("\nLoading Redlich-Kwong Library: %s\n", filename);
}


/*---------------------------------------------------------*/
/* FUNCTION: RKEOS_pressure                                */
/*          Returns density given T and density            */
/*---------------------------------------------------------*/

double RKEOS_pressure(double temp, double density)
{
   double v = 1./density;
   double afun = a0*pow(TCRIT/temp,NRK);
   return rgas*temp/(v-bb)-afun/(v*(v+b0));
}
```

```
/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_spvol                                       */
/*           Returns specific volume given T and P             */
/*-------------------------------------------------------------*/

double RKEOS_spvol(double temp, double press)
{
   double a1,a2,a3;
   double vv,vv1,vv2,vv3;
   double qq,qq3,sqq,rr,tt,dd;
   double afun = a0*pow(TCRIT/temp,NRK);

   a1 = c0-rgas*temp/press;
   a2 = -(bb*b0+rgas*temp*b0/press-afun/press);
   a3 = -afun*bb/press;

   /* Solve cubic equation for specific volume */

   qq = (a1*a1-3.*a2)/9.;
   rr = (2*a1*a1*a1-9.*a1*a2+27.*a3)/54.;
   qq3 = qq*qq*qq;
   dd = qq3-rr*rr;

   /* If dd < 0, then we have one real root */
   /* If dd >= 0, then we have three roots -> choose largest root */

   if (dd < 0.) {
     tt = sqrt(-dd)+pow(fabs(rr),0.333333);
     vv = (tt+qq/tt)-a1/3.;
   } else {
     tt = acos(rr/sqrt(qq3));
     sqq = sqrt(qq);
     vv1 = -2.*sqq*cos(tt/3.)-a1/3.;
     vv2 = -2.*sqq*cos((tt+2.*PI)/3.)-a1/3.;
     vv3 = -2.*sqq*cos((tt+4.*PI)/3.)-a1/3.;
     vv = (vv1 > vv2) ? vv1 : vv2;
     vv = (vv > vv3) ? vv : vv3;
   }

   return vv;
}
```

```
/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_density                                  */
/*           Returns density given T and P                  */
/*----------------------------------------------------------*/

double RKEOS_density(double temp, double press, double yi[])
{
   return 1./RKEOS_spvol(temp, press); /* (Kg/m3) */
}


/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdp                                     */
/*           Returns dv/dp given T and rho                  */
/*----------------------------------------------------------*/

double RKEOS_dvdp(double temp, double density)
{
   double a1,a2,a1p,a2p,a3p,v,press;
   double afun = a0*pow(TCRIT/temp,NRK);

   press = RKEOS_pressure(temp, density);
   v = 1./density;

   a1 = c0-rgas*temp/press;
   a2 = -(bb*b0+rgas*temp*b0/press-afun/press);
   a1p = rgas*temp/(press*press);
   a2p = a1p*b0-afun/(press*press);
   a3p = afun*bb/(press*press);

   return -(a3p+v*(a2p+v*a1p))/(a2+v*(2.*a1+3.*v));
}
```

```
/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdt                                     */
/*           Returns dv/dT given T and rho                  */
/*----------------------------------------------------------*/

double RKEOS_dvdt(double temp, double density)
{
    double a1,a2,dadt,a1t,a2t,a3t,v,press;
    double afun = a0*pow(TCRIT/temp,NRK);

    press = RKEOS_pressure(temp, density);
    v = 1./density;

    dadt = -NRK*afun/temp;
    a1 = c0-rgas*temp/press;
    a2 = -(bb*b0+rgas*temp*b0/press-afun/press);
    a1t = -rgas/press;
    a2t = a1t*b0+dadt/press;
    a3t = -dadt*bb/press;

    return -(a3t+v*(a2t+v*a1t))/(a2+v*(2.*a1+3.*v));
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_Cp_ideal_gas                             */
/*           Returns ideal gas specific heat given T        */
/*----------------------------------------------------------*/

double RKEOS_Cp_ideal_gas(double temp)
{
    return (CC1+temp*(CC2+temp*(CC3+temp*(CC4+temp*CC5))));
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_H_ideal_gas                              */
/*           Returns ideal gas specific enthalpy given T    */
/*----------------------------------------------------------*/

double RKEOS_H_ideal_gas(double temp)
{
    return temp*(CC1+temp*(0.5*CC2+temp*(0.333333*CC3+
              temp*(0.25*CC4+temp*0.2*CC5))));
}
```

```
/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_specific_heat                               */
/*           Returns specific heat given T and rho             */
/*-------------------------------------------------------------*/

double RKEOS_specific_heat(double temp, double density, double P, double yi[])
{
   double delta_Cp,press,v,dvdt,dadt;
   double afun = a0*pow(TCRIT/temp,NRK);

   press = RKEOS_pressure(temp, density);
   v = 1./density;
   dvdt = RKEOS_dvdt(temp, density);
   dadt = -NRK*afun/temp;
   delta_Cp = press*dvdt-rgas-dadt*(1.+NRK)/b0*log((v+b0)/v)
            + afun*(1.+NRK)*dvdt/(v*(v+b0));

   return RKEOS_Cp_ideal_gas(temp)+delta_Cp;  /* (J/Kg-K) */
}

/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy                                    */
/*           Returns specific enthalpy given T and rho         */
/*-------------------------------------------------------------*/

double RKEOS_enthalpy(double temp, double density, double P, double yi[])
{
   double delta_h ,press, v;
   double afun = a0*pow(TCRIT/temp,NRK);

   press = RKEOS_pressure(temp, density);
   v = 1./density;
   delta_h = press*v-rgas*temp-afun*(1+NRK)/b0*log((v+b0)/v);

   return H_REF+RKEOS_H_ideal_gas(temp)+delta_h;  /* (J/Kg) */
}
```

```
/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_entropy                                  */
/*           Returns entropy given T and rho                */
/*----------------------------------------------------------*/

double RKEOS_entropy(double temp, double density, double P, double yi[])
{
   double delta_s,v,v0,dadt,cp_integral;
   double afun = a0*pow(TCRIT/temp,NRK);

   cp_integral = CC1*log(temp)+temp*(CC2+temp*(0.5*CC3+
                 temp*(0.333333*CC4+0.25*CC5*temp)))
                 - cp_int_ref;
   v = 1./density;
   v0 = rgas*temp/P_REF;
   dadt = -NRK*afun/temp;
   delta_s = rgas*log((v-bb)/v0)+dadt/b0*log((v+b0)/v);

   return S_REF+cp_integral+delta_s; /* (J/Kg-K) */
}


/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_mw                                       */
/*           Returns molecular weight                       */
/*----------------------------------------------------------*/

double RKEOS_mw(double yi[])
{
   return MWT; /* (Kg/Kmol) */
}
```

```
/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_speed_of_sound                             */
/*           Returns s.o.s given T and rho                    */
/*------------------------------------------------------------*/

double RKEOS_speed_of_sound(double temp, double density, double P,
double yi[])
{
   double cp = RKEOS_specific_heat(temp, density, P, yi);
   double dvdt = RKEOS_dvdt(temp, density);
   double dvdp = RKEOS_dvdp(temp, density);
   double v = 1./density;
   double delta_c = -temp*dvdt*dvdt/dvdp;

   return sqrt(cp/((delta_c-cp)*dvdp))*v;  /* m/s */
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_rho_t                                      */
/*------------------------------------------------------------*/

double RKEOS_rho_t(double temp, double density, double P, double yi[])
{
   return -density*density*RKEOS_dvdt(temp, density);
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_rho_p                                      */
/*------------------------------------------------------------*/

double RKEOS_rho_p(double temp, double density, double P, double yi[])
{
   return -density*density*RKEOS_dvdp(temp, density);
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy_t                                 */
/*------------------------------------------------------------*/

double RKEOS_enthalpy_t(double temp, double density, double P, double yi[])
{
   return RKEOS_specific_heat(temp, density, P, yi);
}
```

```
/*--------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy_p                                   */
/*--------------------------------------------------------------*/

double RKEOS_enthalpy_p(double temp, double density, double P, double yi[])
{
   double v = 1./density;
   double dvdt = RKEOS_dvdt(temp, density);

   return v-temp*dvdt;
}


/*--------------------------------------------------------------*/
/* FUNCTION: RKEOS_viscosity                                    */
/*--------------------------------------------------------------*/

double RKEOS_viscosity(double temp, double density, double P, double yi[])
{
   double mu,tr,tc,pcatm;

   tr = temp/TCRIT;
   tc = TCRIT;
   pcatm = PCRIT/101325.;

   mu = 6.3e-7*sqrt(MWT)*pow(pcatm,0.6666)/pow(tc,0.16666)*
        (pow(tr,1.5)/(tr+0.8));

   return mu;
}


/*--------------------------------------------------------------*/
/* FUNCTION: RKEOS_thermal_conductivity                         */
/*--------------------------------------------------------------*/

double RKEOS_thermal_conductivity(double temp, double density, double P,
  double yi[])
{
   double cp, mu;

   cp = RKEOS_Cp_ideal_gas(temp);
   mu = RKEOS_viscosity(temp, density, yi);

   return (cp+1.25*rgas)*mu;
```

```
}

/* Export Real Gas Functions to Solver */

UDF_EXPORT RGAS_Functions RealGasFunctionList =
{
   RKEOS_Setup,                  /* initialize          */
   RKEOS_density,                /* density             */
   RKEOS_enthalpy,               /* enthalpy            */
   RKEOS_entropy,                /* entropy             */
   RKEOS_specific_heat,          /* specific_heat       */
   RKEOS_mw,                     /* molecular_weight    */
   RKEOS_speed_of_sound,         /* speed_of_sound      */
   RKEOS_viscosity,              /* viscosity           */
   RKEOS_thermal_conductivity,   /* thermal_conductivity */
   RKEOS_rho_t,                  /* drho/dT |const p    */
   RKEOS_rho_p,                  /* drho/dp |const T    */
   RKEOS_enthalpy_t,             /* dh/dT |const p      */
   RKEOS_enthalpy_p              /* dh/dp |const T      */
};
```

UDRGM Example: Multiple-Species Real Gas Model

This is a simple example for multiple-species real gas model that provide you with a template which you can use to write a more complex multiple-species UDRGM.

In this example a fluid material is defined in the setup function as a mixture of four species (H20, N2, O2, CO2). The equation of state was the simple ideal gas equation of state. The other thermodynamic properties where defined by an ideal-gas mixing law.

Other auxiliary functions are written to provide individual species property to the principle function set.

The example also provide numerical method of computing $\frac{d\rho}{dT}$, $\frac{d\rho}{dp}$, $\frac{dh}{dT}$, and $\frac{dh}{dp}$.

```
/*
 *sccs id: @(#)real_ideal.c 1.10 Copyright 1900/11/09 ANSYS, Inc.
 */

/*
 *      Copyright 1988-1998 ANSYS, Inc.
 *      All Rights Reserved
```

```
 *
 *        This is unpublished proprietary source code of ANSYS, Inc.
 *        It is protected by U.S. copyright law as an unpublished work
 *        and is furnished pursuant to a written license agreement.  It
 *        is considered by ANSYS, Inc. to be confidential and may not be
 *        used, copied, or disclosed to others except in accordance with
 *        the terms and conditions of the license agreement.
 */


/*
 *   Windows Warning!!!  Including udf.h is for getting definitions for
 *                   ANSYS FLUENT constructs such as Domain.   You must
 *                   NOT reference any ANSYS FLUENT globals directly from
 *                   within this module nor link this against any ANSYS
 *                   FLUENT libs, doing so will cause dependencies on a
 *                   specific ANSYS FLUENT binary such as fl551.exe and
 *                   thus won't be version independent.
 */

#include "udf.h"
#include "stdio.h"
#include "ctype.h"
#include "stdarg.h"

#if RP_DOUBLE
#define SMALL 1.e-20
#else
#define SMALL 1.e-10
#endif

#define NCMAX 20
#define NSPECIE_NAME 80


static int  (*usersMessage)(char *,...);
static void (*usersError)(char *,...);



static double ref_p, ref_T;
static char   gas[NCMAX][NSPECIE_NAME];
static int    n_specs;
```

```
double Mixture_Rgas(double yi[]);
double Mixture_pressure(double Temp, double Rho, double yi[]);
double Mw_i(int i);
double Cp_i(double T, double r, int i);
double K_i(double T, double r, int i);
double Mu_i(double T, double r, int i);
double Rgas_i(double T, double r, int i);
double Gm_i(double T, double r, int i);


DEFINE_ON_DEMAND(I_do_nothing)
{
  /*
    This is a dummy function
    must be included to allow for the use of the
    ANSYS FLUENT UDF compilation utility
  */
}

void Mixture_error(int err, char *f, char *msg)
{
  if (err)
    usersError("Mixture_error (%d) from function: %s\n%s\n",err,f,msg);
}

/********************************************************************/
/* Mixture Functions                                             */
/* These are the only functions called from ANSYS FLUENT Code    */
/********************************************************************/
void MIXTURE_Setup(Domain *domain, cxboolean vapor_phase, char *specielist,
                   int  (*messagefunc)(char *format, ...),
                   void (*errorfunc)(char *format, ...))
{
  /* This function will be called from ANSYS FLUENT after the
     UDF library has been loaded.
     User must enter the number of species in the mixture
     and the name of the individual species.
  */

  int i ;
  usersMessage = messagefunc;
  usersError   = errorfunc;
  ref_p        = ABS_P(RP_Get_Real("reference-pressure"),op_pres);
```

```
ref_T        = RP_Get_Real("reference-temperature");

if (ref_p == 0.0)
  {
    Message0("\n MIXTURE_Setup: reference-pressure was not set by user \n");
    Message0("\n MIXTURE_Setup: setting reference-pressure to 101325 Pa \n");
    ref_p = 101325.0 ;
  }
/*======================================================*/
/*=========   User Input Section ====================*/
/*======================================================*/
/*
  Define Number of species & Species name.
  DO NOT use space for naming species
*/
n_specs = 4 ;

(void)strcpy(gas[0],"H2O") ;
(void)strcpy(gas[1],"N2")  ;
(void)strcpy(gas[2],"O2")  ;
(void)strcpy(gas[3],"CO2") ;


/*======================================================*/
/*=========   End Of User Input Section ============*/
/*======================================================*/

Message0("\n MIXTURE_Setup: RealGas mixture initialization \n");
Message0("\n MIXTURE_Setup: Number of Species = %d  \n",n_specs);
for (i=0; i<n_specs; ++i)
  {
    Message0("\n MIXTURE_Setup: Specie[%d]        = %s  \n",i,gas[i]);
  }

/*
  concatenate species name into one string
  and send back to fluent
*/
strcat(specielist,gas[0]);
for (i=1; i<n_specs; ++i)
  {
    strcat(specielist," ");
    strcat(specielist,gas[i]);
  }
```

```
}

double MIXTURE_density(double Temp, double P, double yi[])
{
  double Rgas =  Mixture_Rgas(yi) ;

  double r    =  P/(Rgas*Temp); /* Density at Temp & P */

  return r; /* (Kg/m^3) */
}

double MIXTURE_specific_heat(double Temp, double density, double P,
 double yi[])
{
  double cp=0.0 ;
  int i ;

  for (i=0; i<n_specs; ++i)
    cp += yi[i]*Cp_i(Temp,density,i);

  return cp; /* (J/Kg/K) */
}

double MIXTURE_enthalpy(double Temp, double density, double P, double yi[])
{
  double h=0.0;
  int i ;

  for (i=0; i<n_specs; ++i)
    h  += yi[i]*(Temp*Cp_i(Temp,density,i));

  return h; /* (J/Kg) */
}

double MIXTURE_entropy(double Temp, double density, double P, double yi[])
{
  double s   = 0.0   ;
  double Rgas=0.0 ;

  Rgas = Mixture_Rgas(yi);

  s    = MIXTURE_specific_heat(Temp,density,P,yi)*log(Temp/ref_T) -
         Rgas*log(P/ref_p) ;
```

```
  return s; /* (J/Kg/K) */
}

double MIXTURE_mw(double yi[])
{
  double MW, sum=0.0 ;
  int i ;

  for (i=0; i<n_specs; ++i)
    sum += (yi[i]/Mw_i(i)) ;

  MW = 1.0/MAX(sum,SMALL)        ;

  return MW; /* (Kg/Kmol) */
}

double MIXTURE_speed_of_sound(double Temp, double density, double P,
                              double yi[])
{
  double a, cp, Rgas ;

  cp   = MIXTURE_specific_heat(Temp,density,P,yi) ;
  Rgas = Mixture_Rgas(yi) ;


  a    = sqrt(Rgas*Temp* cp/(cp-Rgas) ) ;

  return a ;  /* m/s */
}

double MIXTURE_viscosity(double Temp, double density, double P, double yi[])
{
  double mu=0;
  int i ;

  for (i=0; i<n_specs; ++i)
    mu += yi[i]*Mu_i(Temp,density,i);

  return mu; /* (Kg/m/s) */
}

double MIXTURE_thermal_conductivity(double Temp, double density, double P,
```

```
                                               double yi[])
{
  double kt=0;
  int i ;

  for (i=0; i<n_specs; ++i)
    kt += yi[i]*K_i(Temp,density,i);

  return kt; /* W/m/K */
}

double MIXTURE_rho_t(double Temp, double density, double P, double yi[])
{

  double drdT   ; /* derivative of rho w.r.t. Temp */
  double p      ;
  double dT=0.01;

  p    = Mixture_pressure(Temp,density, yi);
  drdT = (MIXTURE_density(Temp+dT,p,yi) - MIXTURE_density(Temp,p,yi) ) /dT;

  return drdT; /* (Kg/m^3/K) */
}

double MIXTURE_rho_p(double Temp, double density, double P, double yi[])
{
  double drdp   ;
  double p      ;
  double dp= 5.0 ;

  p    = Mixture_pressure(Temp,density, yi);
  drdp = (MIXTURE_density(Temp,p+dp,yi) - MIXTURE_density(Temp,p,yi) ) /dp;

  return drdp; /* (Kg/m^3/Pa) */
}

double MIXTURE_enthalpy_t(double Temp, double density, double P, double yi[])
{
  double dhdT    ;
  double p       ;
  double rho2    ;
  double dT= 0.01 ;
```

```
    p     = Mixture_pressure(Temp,density, yi);
    rho2  = MIXTURE_density(Temp+dT,p,yi)      ;

    dhdT  = (MIXTURE_enthalpy(Temp+dT,rho2,P,yi) - MIXTURE_enthalpy(Temp,
            density,P,yi)) /dT ;


    return dhdT ; /* J/(Kg.K) */
}

double MIXTURE_enthalpy_p(double Temp, double density, double P, double yi[])
{
  double dhdp     ;
  double p        ;
  double rho2     ;
  double dp= 5.0  ;

    p     = Mixture_pressure(Temp,density, yi);
    rho2  = MIXTURE_density(Temp,p+dp,yi)      ;

    dhdp  = (MIXTURE_enthalpy(Temp,rho2,P,yi) - MIXTURE_enthalpy(Temp,density,
            P,yi)) /dp;


    return dhdp ;  /* J/ (Kg.Pascal) */
}


/******************************************************************/
/* Auxiliary Mixture Functions                                   */
/******************************************************************/


double Mixture_Rgas(double yi[])
{
  double Rgas=0.0 ;
  int i ;

  for (i=0; i<n_specs; ++i)
    Rgas += yi[i]*(UNIVERSAL_GAS_CONSTANT/Mw_i(i)) ;

  return Rgas ;
}
```

```
double Mixture_pressure(double Temp, double Rho, double yi[])
{
  double Rgas =  Mixture_Rgas(yi) ;

  double P    =  Rho*Rgas*Temp    ;  /* Pressure at Temp & P */

  return P; /* (Kg/m^3) */
}




/*******************************************************************/
/* Species Property Functions                                     */
/*******************************************************************/
double Mw_i(int i)
{
  double mi[20];

  mi[0] = 18.01534 ; /*H2O*/
  mi[1] = 28.01340 ; /*N2 */
  mi[2] = 31.99880 ; /*O2 */
  mi[3] = 44.00995 ; /*CO2*/

  return mi[i] ;
}

double Cp_i(double T, double r, int i)
{
  double cpi[20] ;

  cpi[0] = 2014.00 ; /*H2O*/
  cpi[1] = 1040.67 ; /*N2 */
  cpi[2] =  919.31 ; /*O2 */
  cpi[3] =  840.37 ; /*CO2*/

  return cpi[i] ;
}

double K_i(double T, double r, int i)
{
  double ki[20] ;
```

```
    ki[0] = 0.02610 ; /*H2O*/
    ki[1] = 0.02420 ; /*N2 */
    ki[2] = 0.02460 ; /*O2 */
    ki[3] = 0.01450 ; /*CO2*/

    return ki[i] ;
}

double Mu_i(double T, double r, int i)
{
  double mui[20] ;

  mui[0] = 1.340E-05 ; /*H2O*/
  mui[1] = 1.663E-05 ; /*N2 */
  mui[2] = 1.919E-05 ; /*O2 */
  mui[3] = 1.370E-05 ; /*CO2*/

  return mui[i] ;
}

double Rgas_i(double T, double r, int i)
{
  double Rgasi ;

  Rgasi =  UNIVERSAL_GAS_CONSTANT/Mw_i(i) ;

  return Rgasi ;
}

double Gm_i(double T, double r, int i)
{
  double gammai ;

  gammai =  Cp_i(T,r,i)/(Cp_i(T,r,i) - Rgas_i(T,r,i));

  return gammai ;

}


/********************************************************************/
```

```
/* Mixture Functions Structure                                          */
/*********************************************************************/
UDF_EXPORT RGAS_Functions RealGasFunctionList =
{
  MIXTURE_Setup,/* initialize   */
  MIXTURE_density,          /* density              */
  MIXTURE_enthalpy,         /* enthalpy     */
  MIXTURE_entropy,          /* entropy      */
  MIXTURE_specific_heat,/* specific_heat    */
  MIXTURE_mw,          /* molecular_weight       */
  MIXTURE_speed_of_sound,/* speed_of_sound       */
  MIXTURE_viscosity,         /* viscosity    */
  MIXTURE_thermal_conductivity,       /* thermal_conductivity   */
  MIXTURE_rho_t,/* drho/dT |const p     */
  MIXTURE_rho_p,/* drho/dp |const T     */
  MIXTURE_enthalpy_t,          /* dh/dT |const p     */
  MIXTURE_enthalpy_p         /* dh/dp |const T     */
};


/*********************************************************************/
/*********************************************************************/
```

UDRGM Example: Real Gas Model with Volumetric Reactions

This is an example of a UDRGM that has been set up for reacting flow simulations. The example UDF code consists of the following sections:

- Definitions and constants for the physical properties of the species in the single-step methane/air reaction mixture (CH4, O2, N2, CO2, H2O).

- Functions of the Redlich-Kwong equation of state for the individual species property calculations.

- Functions for the mixture properties. In this example, the mixture properties are computed assuming ideal gas mixing rules.

i In the UDRGM only the mixture species and associated properties are defined. No information about the chemical reactions is required in the UDF. The chemical reaction is set up in a separate step, after the UDF has been compiled and loaded into ANSYS FLUENT. See Section 15.1.3: Defining Reactions in the separate User's Guide for details.

```
/*
 *sccs id: @(#)real_ideal.c 1.10 Copyright 1900/11/09 ANSYS, Inc.
 */

/*
 *      Copyright 1988-1998 ANSYS, Inc.
 *      All Rights Reserved
 *
 *      This is unpublished proprietary source code of ANSYS, Inc.
 *      It is protected by U.S. copyright law as an unpublished work
 *      and is furnished pursuant to a written license agreement.  It
 *      is considered by ANSYS, Inc. to be confidential and may not be
 *      used, copied, or disclosed to others except in accordance with
 *      the terms and conditions of the license agreement.
 */


/*
 *    Warning!!!  Including udf.h is for getting definitions for
 *                ANSYS FLUENT constructs such as Domain.   You must
 *                NOT reference any ANSYS FLUENT globals directly from
 *                within this module nor link this against any ANSYS
 *                FLUENT libs, doing so will cause dependencies on a
 *                specific ANSYS FLUENT binary such as fl551.exe and
 *                thus won't be version independent.
 */

#include "udf.h"
#include "stdio.h"
#include "ctype.h"
#include "stdarg.h"

#if RP_DOUBLE
#define SMLL 1.e-20
#else
#define SMLL 1.e-10
#endif

#define NSPECIE_NAME 80
#define RGASU UNIVERSAL_GAS_CONSTANT /* 8314.34 SI units: J/Kmol/K */
#define PI    3.141592654
/* Here input the number of species in the mixture */
/* THIS IS A USER INPUT */
```

```
#define n_specs 5

static int  (*usersMessage)(char *,...);
static void (*usersError)(char *,...);

static double ref_p, ref_T;
static char   gas[n_specs][NSPECIE_NAME];

/* static property parameters */
static double cp[5][n_specs]; /* specific heat polynomial coefficients */
static double mw[n_specs];    /* molecular weights */
static double hf[n_specs];    /* formation enthalpy */
static double tcrit[n_specs]; /* critical temperature */
static double pcrit[n_specs]; /* critical pressure */
static double vcrit[n_specs]; /* critical specific volume */
static double nrk[n_specs];   /* exponent n of function a(T) in Redlich-Kwong
                                 equation of state */
static double omega[n_specs]; /* acentric factor */

/* Static variables associated with Redlich-Kwong Model */
static double rgas[n_specs], a0[n_specs], b0[n_specs], c0[n_specs],
  bb[n_specs], cp_int_ref[n_specs];

void  Mw();
void  Cp_Parameters();
void  Hform();
void  Tcrit();
void  Pcrit();
void  Vcrit();
void  NRK();
void Omega();

double RKEOS_spvol(double temp, double press, int i);
double RKEOS_dvdp(double temp, double density, int i);
double RKEOS_dvdt(double temp, double density, int i);
double RKEOS_H_ideal_gas(double temp, int i);
double RKEOS_specific_heat(double temp, double density, int i);
double RKEOS_enthalpy(double temp, double density, int i);
double RKEOS_entropy(double temp, double density, int i);
double RKEOS_viscosity(double temp, int i);
double RKEOS_thermal_conductivity(double temp, int i);
double RKEOS_vol_specific_heat(double temp, double density, int i);
```

```
DEFINE_ON_DEMAND(I_do_nothing)
{
  /*
    This is a dummy function
    must be included to allow for the use of the
    ANSYS FLUENT UDF compilation utility
  */
}

void Mixture_error(int err, char *f, char *msg)
{
  if (err)
    usersError("Mixture_error (%d) from function: %s\n%s\n",err,f,msg);
}

/**********************************************************************/
/* Mixture Functions                                                  */
/* These are the only functions called from ANSYS FLUENT Code         */
/**********************************************************************/
void MIXTURE_Setup(Domain *domain, cxboolean vapor_phase, char *specielist,
                   int  (*messagefunc)(char *format, ...),
                   void (*errorfunc)(char *format, ...))
{
  /* This function will be called from ANSYS FLUENT after the
     UDF library has been loaded.
     User must enter the number of species in the mixture
     and the name of the individual species.
  */

  int i ;
  usersMessage = messagefunc;
  usersError   = errorfunc;
  ref_p        = ABS_P(RP_Get_Real("reference-pressure"),op_pres);
  ref_T        = 298.15 ;

  Message0("\n MIXTURE_Setup: Redlich-Kwong equation of State"
           " with ideal-gas mixing rules \n");
  Message0("\n MIXTURE_Setup: reference-temperature is %f \n", ref_T);

  if (ref_p == 0.0)
    {
      Message0("\n MIXTURE_Setup: reference-pressure was not set by user \n");
      Message0("\n MIXTURE_Setup: setting reference-pressure to 101325 Pa \n");
```

```
      ref_p = 101325.0 ;
  }
/*========================================================*/
/*=========    User Input Section ====================*/
/*========================================================*/
/*
  Define Species name.
  DO NOT use space for naming species
*/
(void)strcpy(gas[0],"H2O") ;
(void)strcpy(gas[1],"CH4") ;
(void)strcpy(gas[2],"O2")  ;
(void)strcpy(gas[3],"CO2") ;
(void)strcpy(gas[4],"N2")  ;


/*========================================================*/
/*=========    End Of User Input Section =============*/
/*========================================================*/

Message0("\n MIXTURE_Setup: RealGas mixture initialization \n");
Message0("\n MIXTURE_Setup: Number of Species = %d  \n",n_specs);
for (i=0; i<n_specs; ++i)
  {
    Message0("\n MIXTURE_Setup: Specie[%d]        = %s  \n",i,gas[i]);
  }

/*
  concatenate species name into one string
  and send back to fluent
*/
strcat(specielist,gas[0]);
for (i=1; i<n_specs; ++i)
  {
    strcat(specielist," ");
    strcat(specielist,gas[i]);
  }

/* initialize */
Mw();
Cp_Parameters();
Hform();
Tcrit();
Pcrit();
```

```
    Vcrit();
    Omega();
    NRK();

    for (i=0; i<n_specs; ++i)
      {
        rgas[i] = RGASU/mw[i];
        a0[i] = 0.42747*rgas[i]*rgas[i]*tcrit[i]*tcrit[i]/pcrit[i];
        b0[i] = 0.08664*rgas[i]*tcrit[i]/pcrit[i];
        c0[i] = rgas[i]*tcrit[i]/(pcrit[i]+a0[i]/(vcrit[i]*(vcrit[i]+b0[i])))
                +b0[i]-vcrit[i];
        bb[i] = b0[i]-c0[i];
        cp_int_ref[i] = cp[0][i]*log(ref_T)+ref_T*(cp[1][i]+ref_T*(0.5*cp[2][i]
                        +ref_T*(0.333333*cp[3][i]+0.25*cp[4][i]*ref_T)));
      }
}


double MIXTURE_mw(double yi[])
{
  double MW, sum=0.0 ;
  int i ;

  for (i=0; i<n_specs; ++i)
    sum += yi[i]/mw[i] ;

  MW = 1.0/MAX(sum,SMLL)      ;

  return MW; /* (Kg/Kmol) */
}

double MIXTURE_density(double temp, double P, double yi[])
{
  double den=0.0 ;
  int i ;

  for (i=0; i<n_specs; ++i)
    {
      if (yi[i]> SMLL)
   den += yi[i]*RKEOS_spvol(temp, P, i);
    }

  return 1./den; /* (Kg/m^3) */
}
```

```
double MIXTURE_specific_heat(double temp, double density, double P,
                             double yi[])
{
  double cp=0.0 ;
  int i ;

  for (i=0; i<n_specs; ++i)
      if (yi[i]> SMLL)
    cp += yi[i]*RKEOS_specific_heat(temp,mw[i]*density/MIXTURE_mw(yi),i);

  return cp; /* (J/Kg/K) */
}


double MIXTURE_enthalpy(double temp, double density, double P, double yi[])
{
  double h=0.0;
  int i ;

  for (i=0; i<n_specs; ++i)
      if (yi[i]> SMLL)
    h  += yi[i]*RKEOS_enthalpy(temp, mw[i]*density/MIXTURE_mw(yi), i);

  return h; /* (J/Kg) */
}


double MIXTURE_enthalpy_prime(double temp, double density, double P,
                             double yi[], double hi[])
{
  double h=0.0;
  int i ;

  for (i=0; i<n_specs; ++i)
{
    hi[i] = hf[i]/mw[i] + RKEOS_enthalpy(temp, mw[i]*density/MIXTURE_mw(yi),
            i);
      if (yi[i]> SMLL)
    h  += yi[i]*(hf[i]/mw[i] + RKEOS_enthalpy(temp,
          mw[i]*density/MIXTURE_mw(yi), i));
}

  return h; /* (J/Kg) */
}
```

```
double MIXTURE_entropy(double temp, double density, double P, double yi[])
{
  double s   = 0.0   ;
  double sum = 0.0 ;
  double xi[n_specs] ;
  int i ;

  for (i=0; i<n_specs; ++i)
    {
      xi[i] = yi[i] / mw[i];
      sum   += xi[i];
    }
  for (i=0; i<n_specs; ++i)
    xi[i] /= sum;

  for (i=0; i<n_specs; ++i)
      if (yi[i]> SMLL)
    s  += yi[i]*RKEOS_entropy(temp,mw[i]*density/MIXTURE_mw(yi), i)-
        UNIVERSAL_GAS_CONSTANT/MIXTURE_mw(yi)* xi[i] * log(xi[i]);

  return s; /* (J/Kg/K) */
}

double MIXTURE_viscosity(double temp, double density, double P, double yi[])
{
  double mu=0.;
  int i ;

  for (i=0; i<n_specs; ++i)
      if (yi[i]> SMLL)
    mu += yi[i]*RKEOS_viscosity(temp,i);

  return mu; /* (Kg/m/s) */
}

double MIXTURE_thermal_conductivity(double temp, double density, double P,
                                    double yi[])
{
  double kt=0.;
  int i ;

  for (i=0; i<n_specs; ++i)
```

```
      if (yi[i]> SMLL)
    kt += yi[i]* RKEOS_thermal_conductivity(temp,i);

  return kt; /* W/m/K */
}


/*---------------------------------------------------------*/
/* FUNCTION: MIXTURE_speed_of_sound                        */
/*           Returns s.o.s given T and rho                 */
/*---------------------------------------------------------*/

double MIXTURE_speed_of_sound(double temp, double density, double P,
                              double yi[])
{
   double dvdp = 0.;
   double cv = 0.;
   double v = 1./density;
   int i;
   double cp = MIXTURE_specific_heat(temp, density, P, yi);

   for (i=0; i<n_specs; ++i)
     if (yi[i]> SMLL)
    {
      dvdp += yi[i]*RKEOS_dvdp(temp, mw[i]*density/MIXTURE_mw(yi),i);
      cv += yi[i]*RKEOS_vol_specific_heat(temp, mw[i]*density/MIXTURE_mw(yi),
           i);
    }

   return sqrt(- cp/cv/dvdp)*v;
}


/*---------------------------------------------------------*/
/* FUNCTION: MIXTURE_rho_t                                 */
/*---------------------------------------------------------*/

double MIXTURE_rho_t(double temp, double density, double P, double yi[])
{
  double rho_t = 0.;
  int i;

  for (i=0; i<n_specs; ++i)
      if (yi[i]> SMLL)
   rho_t -= yi[i]*density*density*RKEOS_dvdt(temp,
```

```
         mw[i]*density/MIXTURE_mw(yi) , i);
   return rho_t;
 }


 /*-----------------------------------------------------------*/
 /* FUNCTION: MIXTURE_rho_p                                   */
 /*-----------------------------------------------------------*/

 double MIXTURE_rho_p(double temp, double density, double P, double yi[])
 {
   double rho_p = 0.;
   int i;

   for (i=0; i<n_specs; ++i)
       if (yi[i]> SMLL)
    rho_p -= yi[i]*density*density*RKEOS_dvdp(temp,
            mw[i]*density/MIXTURE_mw(yi), i);
   return rho_p;
 }


 /*-----------------------------------------------------------*/
 /* FUNCTION: MIXTURE_enthalpy_t                              */
 /*-----------------------------------------------------------*/

 double MIXTURE_enthalpy_t(double temp, double density, double P, double yi[])
 {
    return MIXTURE_specific_heat(temp, density, P, yi);
 }


 /*-----------------------------------------------------------*/
 /* FUNCTION: MIXTURE_enthalpy_p                              */
 /*-----------------------------------------------------------*/

 double MIXTURE_enthalpy_p(double temp, double density, double P, double yi[])
 {
    double v = 1./density;
    double dvdt = 0.0;
    int i;

    for (i=0; i<n_specs; ++i)
       if (yi[i]> SMLL)
       dvdt += yi[i]*RKEOS_dvdt(temp,  mw[i]*density/MIXTURE_mw(yi), i);
```

```
   return v-temp*dvdt;
}


/**********************************************************************/
/* Species Property Definitions                                      */
/**********************************************************************/

void Mw() /* molecular weight */
{  /* Kg/Kgmol */
  mw[0] = 18.01534 ; /*H2O*/
  mw[1] = 16.04303 ; /*CH4*/
  mw[2] = 31.99880 ; /*O2 */
  mw[3] = 44.00995 ; /*CO2*/
  mw[4] = 28.01340 ; /*N2 */
}

void Pcrit() /* critical pressure */
{ /* Pa */
  pcrit[0] = 220.64e5 ; /*H2O*/
  pcrit[1] = 4.48e6 ; /*CH4*/
  pcrit[2] = 5066250.; /*O2 */
  pcrit[3] = 7.3834e6 ; /*CO2*/
  pcrit[4] = 33.98e5 ; /*N2 */
}

void Tcrit() /* critical temperature */
{  /* K */
  tcrit[0] = 647. ; /*H2O*/
  tcrit[1] = 191. ; /*CH4*/
  tcrit[2] = 155.; /*O2 */
  tcrit[3] = 304.; /*CO2*/
  tcrit[4] = 126.2 ; /*N2 */
}

void Vcrit() /* critical specific volume */
{ /* m3/Kg */
  vcrit[0] = 0.003111 ; /*H2O*/
  vcrit[1] = 0.006187 ; /*CH4*/
  vcrit[2] = 0.002294 ; /*O2 */
  vcrit[3] = 0.002136; /*CO2*/
  vcrit[4] = 0.003196; /*N2 */
}
```

```c
void NRK() /* exponent n of function a(T) in Redlich-Kwong equation of
             state */
{
  int i;
  for (i=0; i<n_specs; ++i)
    nrk[i]= 0.4986 + 1.1735*omega[i] + 0.475*omega[i]*omega[i];
}
void Omega() /* acentric factor */
{
  omega[0] = 0.348 ; /*H2O*/
  omega[1] = 0.007; /*CH4*/
  omega[2] = 0.021 ; /*O2 */
  omega[3] = 0.225; /*CO2*/
  omega[4] = 0.040; /*N2 */
}


void Hform() /* formation enthalpy */
{
  /*J/Kgmol*/
  hf[0] =  -2.418379e+08; /*H2O*/
  hf[1] =  -74895176. ;   /*CH4*/
  hf[2] = 0. ;            /*O2 */
  hf[3] = -3.9353235e+08 ;/*CO2*/
  hf[4] = 0. ;            /*N2 */
}

void Cp_Parameters( ) /* coefficients of specific heat polynomials */
{  /* J/Kg/K */
  cp[0][0] =  1609.791  ; /*H2O*/
  cp[1][0] = 0.740494;
  cp[2][0] =-9.129835e-06 ;
  cp[3][0] =-3.813924e-08 ;
  cp[4][0] =4.80227e-12 ;

  cp[0][1] = 872.4671  ; /*CH4*/
  cp[1][1] = 5.305473 ;
  cp[2][1] = -0.002008295 ;
  cp[3][1] = 3.516646e-07;
  cp[4][1] = -2.33391e-11  ;

  cp[0][2] = 811.1803  ; /*O2 */
  cp[1][2] =0.4108345 ;
  cp[2][2] =-0.0001750725 ;
```

```
    cp[3][2] = 3.757596e-08 ;
    cp[4][2] =-2.973548e-12 ;

    cp[0][3] = 453.577 ; /*CO2*/
    cp[1][3] = 1.65014;
    cp[2][3] = -1.24814e-3 ;
    cp[3][3] = 3.78201e-7 ;
    cp[4][3] = 0.;

    cp[0][4] = 938.8992 ; /*N2 */
    cp[1][4] = 0.3017911 ;
    cp[2][4] = -8.109228e-05;
    cp[3][4] =  8.263892e-09  ;
    cp[4][4] = -1.537235e-13 ;
}


/***************************************************************/
/*                                                           */
/* User-Defined Function: Redlich-Kwong Equation of State    */
/*                        for Real Gas Modeling              */
/*                                                           */
/*  Author: Frank Kelecy                                     */
/*    Date: May 2003                                         */
/*Modified: Rana Faltsi                                      */
/*    Date: December 2006                                    */
/*                                                           */
/*                                                           */
/***************************************************************/
/* OPTIONAL REFERENCE (OFFSET) VALUES FOR ENTHALPY AND ENTROPY */

#define H_REF  0.0
#define S_REF  0.0
/*-----------------------------------------------------------*/
/* FUNCTION: RKEOS_pressure of species i                     */
/*           Returns pressure given T and density            */
/*-----------------------------------------------------------*/

double RKEOS_pressure(double temp, double density, int i)
{
   double v = 1./density;
   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
   return rgas[i]*temp/(v-bb[i])-afun/(v*(v+b0[i]));
}
```

```
/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_spvol of species i                         */
/*           Returns specific volume given T and P            */
/*------------------------------------------------------------*/

double RKEOS_spvol(double temp, double press, int i)
{
   double a1,a2,a3;
   double vv,vv1,vv2,vv3;
   double qq,qq3,sqq,rr,tt,dd;

   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

   a1 = c0[i]-rgas[i]*temp/press;
   a2 = -(bb[i]*b0[i]+rgas[i]*temp*b0[i]/press-afun/press);
   a3 = -afun*bb[i]/press;

   /* Solve cubic equation for specific volume */

   qq = (a1*a1-3.*a2)/9.;
   rr = (2*a1*a1*a1-9.*a1*a2+27.*a3)/54.;
   qq3 = qq*qq*qq;
   dd = qq3-rr*rr;

   /* If dd < 0, then we have one real root */
   /* If dd >= 0, then we have three roots -> choose largest root */

   if (dd < 0.) {
     tt = -SIGN(rr)*(pow(sqrt(-dd)+fabs(rr),0.333333));
     vv = (tt+qq/tt)-a1/3.;
   } else {
     if (rr/sqrt(qq3)<-1) {
       tt = PI;
     } else if (rr/sqrt(qq3)>1) {
       tt = 0;
     } else {
       tt = acos(rr/sqrt(qq3));
     }
     sqq = sqrt(qq);
     vv1 = -2.*sqq*cos(tt/3.)-a1/3.;
     vv2 = -2.*sqq*cos((tt+2.*PI)/3.)-a1/3.;
     vv3 = -2.*sqq*cos((tt+4.*PI)/3.)-a1/3.;
```

```
      vv = (vv1 > vv2) ? vv1 : vv2;
      vv = (vv > vv3) ? vv : vv3;
       /*Message0("Three roots %f %f %f \n",vv1, vv2, vv3 );*/
   }


   return vv;
}


/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdp                                        */
/*           Returns dv/dp given T and rho                     */
/*-------------------------------------------------------------*/

double RKEOS_dvdp(double temp, double density, int i)
{
   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
   double dterm1,dterm2;
   double v     =  1./ density;

   dterm1   = -rgas[i]*temp*pow((v-b0[i]+c0[i]), -2.0);
   dterm2   = afun*(2.0*v+b0[i])*pow(v*(v+b0[i]),-2.0);

   return 1./( dterm1+dterm2);
}


/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdt                                        */
/*           Returns dv/dT given T and rho                     */
/*-------------------------------------------------------------*/

double RKEOS_dvdt(double temp, double density, int i)
{
   double dpdT, dterm1, dterm2;
   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
   double v     = 1./density;

   dterm1   = rgas[i]/(v-b0[i]+c0[i]);
   dterm2   = nrk[i]*afun/((v*(v+b0[i]))*temp);
   dpdT = dterm1+dterm2;

   return  - RKEOS_dvdp(temp, density, i)* dpdT;
}
```

```
/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_Cp_ideal_gas                               */
/*           Returns ideal gas specific heat given T          */
/*------------------------------------------------------------*/

double RKEOS_Cp_ideal_gas(double temp, int i)
{
  double cpi=(cp[0][i]+temp*(cp[1][i]+temp*(cp[2][i]+temp*(cp[3][i]
             +temp*cp[4][i])))); 
  if (cpi<SMLL)
    cpi = 1.0;
  return cpi;
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_H_ideal_gas                                */
/*           Returns ideal gas specific enthalpy given T      */
/*------------------------------------------------------------*/

double RKEOS_H_ideal_gas(double temp, int i)
{
  double h = temp*(cp[0][i]+temp*(0.5*cp[1][i]+temp*(0.333333*cp[2][i]
             +temp*(0.25*cp[3][i]+temp*0.2*cp[4][i])))); 
  if (h<SMLL)
    h = 1.0;
  return h;
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_vol_specific_heat                          */
/*           Returns constant volume specific heat given T and rho */
/*------------------------------------------------------------*/

double RKEOS_vol_specific_heat(double temp, double density, int i)
{
   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
   double v = 1./density;
   double Cv0    = RKEOS_Cp_ideal_gas(temp, i) - rgas[i];
   int np1 = (nrk[i]+1.)/b0[i];
   if (Cv0<SMLL)
     Cv0 = 1.;
   return Cv0 + nrk[i]*np1*afun*log(1.0+b0[i]/v)/temp;
```

```
}
/*---------------------------------------------------------------*/
/* FUNCTION: RKEOS_specific_heat                                 */
/*           Returns specific heat given T and rho               */
/*---------------------------------------------------------------*/

double RKEOS_specific_heat(double temp, double density, int i)
{
   double delta_Cp,press,v,dvdt,dadt;
   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

   press = RKEOS_pressure(temp, density, i);
   v = 1./density;
   dvdt = RKEOS_dvdt(temp, density, i);
   dadt = -nrk[i]*afun/temp;
   delta_Cp = press*dvdt-rgas[i]-dadt*(1.+nrk[i])/b0[i]*log((v+b0[i])/v)
            + afun*(1.+nrk[i])*dvdt/(v*(v+b0[i]));

   return RKEOS_Cp_ideal_gas(temp, i)+delta_Cp;  /* (J/Kg-K) */

}

/*---------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy                                      */
/*           Returns specific enthalpy given T and rho           */
/*---------------------------------------------------------------*/

double RKEOS_enthalpy(double temp, double density, int i)
{
   double delta_h ,press, v;

   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

   press = RKEOS_pressure(temp, density, i);
   v = 1./density;
   delta_h = press*v-rgas[i]*temp-afun*(1+nrk[i])/b0[i]*log((v+b0[i])/v);

   return H_REF+RKEOS_H_ideal_gas(temp,i)+delta_h;  /* (J/Kg) */
}

/*---------------------------------------------------------------*/
/* FUNCTION: RKEOS_entropy                                       */
/*           Returns entropy given T and rho                     */
```

```
/*-----------------------------------------------------------*/

double RKEOS_entropy(double temp, double density, int i)
{
   double delta_s,v,v0,dadt,cp_integral;

   double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

   cp_integral = cp[0][i]*log(temp)+temp*(cp[1][i]+temp*(0.5*cp[2][i]
                 +temp*(0.333333*cp[3][i]+0.25*cp[4][i]*temp)))
                - cp_int_ref[i];
   if (cp_integral<SMLL)
     cp_integral = 1.0;
   v = 1./density;
   v0 = rgas[i]*temp/ref_p;
   dadt = -nrk[i]*afun/temp;
   delta_s = rgas[i]*log((v-bb[i])/v0)+dadt/b0[i]*log((v+b0[i])/v);

   return S_REF+cp_integral+delta_s; /* (J/Kg-K) */
}

/*-----------------------------------------------------------*/
/* FUNCTION: RKEOS_viscosity                                 */
/*-----------------------------------------------------------*/

double RKEOS_viscosity(double temp, int i)
{
   double mu,tr,tc,pcatm;

   tr = temp/tcrit[i];
   tc = tcrit[i];
   pcatm = pcrit[i]/101325.;

   mu = 6.3e-7*sqrt(mw[i])*pow(pcatm,0.6666)/pow(tc,0.16666)
        *(pow(tr,1.5)/(tr+0.8));

   return mu;
}

/*-----------------------------------------------------------*/
/* FUNCTION: RKEOS_thermal_conductivity                      */
/*-----------------------------------------------------------*/
```

```
double RKEOS_thermal_conductivity(double temp,int i)
{
   double cp, mu;

   cp = RKEOS_Cp_ideal_gas(temp, i);
   mu = RKEOS_viscosity(temp, i);

   return (cp+1.25*rgas[i])*mu;
}


/*********************************************************************/
/* Mixture Functions Structure                                      */
/*********************************************************************/
UDF_EXPORT RGAS_Functions RealGasFunctionList =
{
  MIXTURE_Setup,/* initialize   */
  MIXTURE_density,        /* density          */
  MIXTURE_enthalpy,        /* sensible enthalpy      */
  MIXTURE_entropy,        /* entropy     */
  MIXTURE_specific_heat,/* specific_heat   */
  MIXTURE_mw,         /* molecular_weight     */
  MIXTURE_speed_of_sound,/* speed_of_sound     */
  MIXTURE_viscosity,        /* viscosity   */
  MIXTURE_thermal_conductivity,      /* thermal_conductivity   */
  MIXTURE_rho_t,/* drho/dT |const p    */
  MIXTURE_rho_p,/* drho/dp |const T    */
  MIXTURE_enthalpy_t,          /* dh/dT |const p    */
  MIXTURE_enthalpy_p,        /* dh/dp |const T    */
  MIXTURE_enthalpy_prime /* enthalpy             */
};
```

Appendix A.   C Programming Basics

This chapter contains an overview of C programming basics for UDFs.

- Section A.1: Introduction

- Section A.2: Commenting Your C Code

- Section A.3: C Data Types in ANSYS FLUENT

- Section A.4: Constants

- Section A.5: Variables

- Section A.6: User-Defined Data Types

- Section A.7: Casting

- Section A.8: Functions

- Section A.9: Arrays

- Section A.10: Pointers

- Section A.11: Control Statements

- Section A.12: Common C Operators

- Section A.13: C Library Functions

- Section A.14: Macro Substitution Directive Using `#define`

- Section A.14: File Inclusion Directive Using `#include`

- Section A.15: Comparison with FORTRAN

A.1   Introduction

This chapter contains some basic information about the C programming language that may be helpful when writing UDFs in ANSYS FLUENT. It is not intended to be used as a primer on C and assumes that you are an experienced programmer in C. There are many topics and details that are *not* covered in this chapter including, for example, while and do-while control statements, unions, recursion, structures, and reading and writing files.

If you are unfamiliar with C, please consult a C language reference guide (e.g., [6, 9]) before you begin the process of writing UDFs for your ANSYS FLUENT model.

A.2　Commenting Your C Code

It is good programming practice to document your C code with comments that are useful for explaining the purpose of the function. In a single line of code, your comments must begin with the `/*` identifier, followed by text, and end with the `*/` identifier as shown by the following:

```
/* This is how I put a comment in my C program  */
```

Comments that span multiple lines are bracketed by the same identifiers:

```
/* This is how I put a comment in my C program
   that spans more
   than one line.      */
```

`i` Do not include a `DEFINE` macro name (e.g., `DEFINE_PROFILE`) within a comment in your source code. This will cause a compilation error.

A.3　C Data Types in ANSYS FLUENT

The UDF interpreter in ANSYS FLUENT supports the following standard C data types:

`int`	integer number
`long`	integer number of increased range
`float`	floating point (real) number
`double`	double-precision floating point (real) number
`char`	single byte of memory, enough to hold a character

Note that in ANSYS FLUENT, `real` is a typedef that switches between `float` for single-precision arithmetic, and `double` for double-precision arithmetic. Since the interpreter makes this assignment automatically, it is good programming practice to use the `real` typedef when declaring all `float` and `double` data type variables in your UDF.

A.4   Constants

Constants are absolute values that are used in expressions and need to be defined in your C program using `#define`. Simple constants are decimal integers (e.g., 0, 1, 2). Constants that contain decimal points or the letter `e` are taken as floating point constants. As a convention, constants are typically declared using all capitals. For example, you may set the ID of a zone, or define constants `YMIN` and `YMAX` as shown below:

```
#define WALL_ID 5
#define YMIN 0.0
#define YMAX 0.4064
```

A.5   Variables

A variable (or object) is a place in memory where you can store a value. Every variable has a type (e.g., `real`), a name, and a value, and may have a storage class identifier (`static` or `extern`). All variables must be declared before they can be used. By declaring a variable ahead of time, the C compiler knows what kind of storage to allocate for the value.

Global variables are variables that are defined outside of any single function and are visible to all function(s) within a UDF source file. Global variables can also be used by other functions outside of the source file unless they are declared as `static` (see Section A.5.3: Static Variables). Global variables are typically declared at the beginning of a file, after preprocessor directives as in

```
#include "udf.h"

real volume;  /* real variable named volume is declared globally */

DEFINE_ADJUST(compute_volume, domain)
{
   /* code that computes volume of some zone  */
   volume = ....
}
```

Local variables are variables that are used in a single function. They are created when the function is called, and are destroyed when the function returns unless they are declared as `static` (see Section A.5.3: Static Variables). Local variables are declared within the body of a function (inside the curly braces `{}`). In the example below, `mu_lam` and `temp` are local variables. The value of these variables is not preserved after the function returns.

```
DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
  real mu_lam;                      /* local variable  */
  real temp = C_T(cell, thread);    /* local variable  */

  if (temp > 288.)
    mu_lam = 5.5e-3;
  else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
  else
    mu_lam = 1.;

  return mu_lam;
}
```

A.5.1 Declaring Variables

A variable declaration begins with the data type (e.g., `int`), followed by the name of one or more variables of the same type that are separated by commas. A variable declaration can also contain an initial value, and always ends with a semicolon (`;`). Variable names must begin with a letter in C. A name can include letters, numbers, and the underscore (`_`) character. Note that the C preprocessor is case-sensitive (recognizes uppercase and lowercase letters as being different). Below are some examples of variable declarations.

```
int n;                  /* declaring variable n as an integer       */
int i1, i2;             /* declaring variables i1 and i2 as integers */
float tmax = 0.;        /* tmax is a floating point real number
                           that is initialized to 0                 */
real average_temp = 0.0; /* average_temp is a real number initialized
                           to 0.0                                   */
```

A.5.2 External Variables

If you have a global variable that is declared in one source code file, but a function in another source file needs to use it, then it must be defined in the other source file as an external variable. To do this, simply precede the variable declaration by the word `extern` as in

```
extern real volume;
```

If there are several files referring to that variable then it is convenient to include the `extern` definition in a header (`.h`) file, and include the header file in all of the `.c` files

that want to use the external variable. Only one `.c` file should have the declaration of the variable without the `extern` keyword. Below is an example that demonstrates the use of a header file.

> *i*   `extern` can be used only in compiled UDFs.

Example

Suppose that there is a global variable named `volume` that is declared in a C source file named `file1.c`

```
#include "udf.h"
real volume;      /* real variable named volume is declared globally */

DEFINE_ADJUST(compute_volume, domain)
{
   /* code that computes volume of some zone  */
   volume = ....
}
```

If multiple source files want to use `volume` in their computations, then `volume` can be declared as an external variable in a header file (e.g., `extfile.h`)

```
/* extfile.h
   Header file that contains the external variable declaration for
   volume  */

extern real volume;
```

Now another file named `file2.c` can declare `volume` as an external variable by simply including `extfile.h`.

```
/* file2.c

#include "udf.h"
#include "extfile.h"  /* header file containing extern declaration
                         is included */

DEFINE_SOURCE(heat_source,c,t,ds,eqn)
{
   /* code that computes the per unit volume source using the total
```

```
      volume computed in the compute_volume function from file1.c   */

   real total_source = ...;
   real source;

   source = total_source/volume;
   return source;
}
```

A.5.3   Static Variables

The `static` operator has different effects depending on whether it is applied to local or global variables. When a local variable is declared as `static` the variable is prevented from being destroyed when a function returns from a call. In other words, the value of the variable is preserved. When a global variable is declared as `static` the variable is "file global". It can be used by any function within the source file in which it is declared, but is prevented from being used outside the file, even if is declared as external. Functions can also be declared as `static`. A static function is visible only to the source file in which it is defined.

> *i* `static` variables and functions can be declared *only* in compiled UDF source files.

Example - Static Global Variable

```
/*  mysource.c  /*

#include "udf.h"

static real abs_coeff = 1.0;  /* static global variable */
    /* used by both functions in this source file but is
        not visible to the outside  */

DEFINE_SOURCE(energy_source, c, t, dS, eqn)
{
 real source;    /* local variable
 int P1 = ....;  /* local variable
                   value is not preserved when function returns */

 dS[eqn] = -16.* abs_coeff * SIGMA_SBC * pow(C_T(c,t),3.);
 source =-abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
 return source;
}
```

```
DEFINE_SOURCE(p1_source, c, t, dS, eqn)
{
 real source;
 int P1 = ...;
 dS[eqn] = -abs_coeff;
 source = abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
 return source;
}
```

A.6   User-Defined Data Types

C also allows you to create user-defined data types using structures and `typedef`. (For information about structures in C, see [6].) An example of a structured list definition is shown below.

> *i*   `typedef` can only be used for compiled UDFs.

Example

```
typedef struct list{int a;
                     real b;
                     int c;} mylist; /* mylist is type structure list
mylist x,y,z;                        x,y,z are type structure list */
```

A.7   Casting

You can convert from one data type to another by casting. A cast is denoted by type, where the type is `int`, `float`, etc., as shown in the following example:

```
int x = 1;
real y = 3.14159;
int z = x+((int) y);       /* z = 4 */
```

A.8   Functions

Functions perform tasks. Tasks may be useful to other functions defined within the same source code file, or they may be used by a function external to the source file. A function has a name (that you supply) and a list of zero or more arguments that are passed to it. Note that your function name cannot contain a number in the first couple of characters. A function has a body enclosed within curly braces that contains instructions for carrying

out the task. A function may return a value of a particular type. C functions pass data by value.

Functions either return a value of a particular data type (e.g., `real`), or do not return any value if they are of type `void`. To determine the return data type for the `DEFINE` macro you will use to define your UDF, look at the macro's corresponding `#define` statement in the `udf.h` file or see Appendix B for a listing.

> i C functions cannot alter their arguments. They can, however, alter the variables that their arguments point to.

A.9 Arrays

Arrays of variables can be defined using the notation `name[size]`, where `name` is the variable name and `size` is an integer that defines the number of elements in the array. The index of a C array always begins at 0.

Arrays of variables can be of different data types as shown below.

Examples

```
int a[10], b[10][10];
real radii[5];

a[0] = 1;                  /* a 1-Dimensional array of variable a     */
radii[4] = 3.14159265;     /* a 1-Dimensional array of variable radii */
b[10][10] = 4;             /* a 2-Dimensional array of variable b     */
```

A.10 Pointers

A pointer is a variable that contains an address in memory where the value referenced by the pointer is stored. In other words, a pointer is a variable that points to another variable by referring to the other variable's address. Pointers contain memory addresses, not values. Pointer variables must be declared in C using the `*` notation. Pointers are widely used to reference data stored in structures and to pass data among functions (by passing the addresses of the data).

For example,

```
int *ip;
```

declares a pointer named `ip` that points to an integer variable.

Now suppose you want to assign an address to pointer `ip`. To do this, you can use the `&` notation. For example,

```
ip = &a;
```

assigns the address of variable `a` to pointer `ip`.

You can retrieve the value of variable `a` that pointer `ip` is pointing to by

```
*ip
```

Alternatively, you can set the value of the variable that pointer `ip` points. For example,

```
*ip = 4;
```

assigns a value of `4` to the variable that pointer `ip` is pointing. The use of pointers is demonstrated by the following:

```
int a = 1;
int *ip;
ip = &a;                    /* &a returns the address of variable a */
printf("content of address pointed to by ip = %d\n", *ip);
*ip = 4;                    /* a = 4  */
printf("now a = %d\n", a);
```

Here, an integer variable `a` is initialized to `1`. Next, `ip` is declared as a pointer to an integer variable. The address of variable `a` is then assigned to pointer `ip`. Next, the integer value of the address pointed to by `ip` is printed using `*ip`. (This value is `1`.) The value of variable `a` is then indirectly set to `4` using `*ip`. The new value of `a` is then printed. Pointers can also point to the beginning of an array, and are strongly connected to arrays in C.

Pointers as Function Arguments

C functions can access and modify their arguments through pointers. In ANSYS FLUENT, thread and domain pointers are common arguments to UDFs. When you specify these arguments in your UDF, the ANSYS FLUENT solver automatically passes data that the pointers are referencing to your UDF so that your function can access solver data. (You do not have to declare pointers that are passed as arguments to your UDF from the solver.) For example, one of the arguments passed to a UDF that specifies a custom profile (defined by the DEFINE_PROFILE macro) is the pointer to the thread applied to by the boundary condition. The DEFINE_PROFILE function accesses the data pointed to by the thread pointer.

A.11   Control Statements

You can control the order in which statements are executed in your C program using control statements like `if`, `if-else`, and `for` loops. Control statements make decisions about what is to be executed next in the program sequence.

A.11.1   `if` **Statement**

An `if` statement is a type of conditional control statement. The format of an `if` statement is:

```
if (logical-expression)
    {statements}
```

where `logical-expression` is the condition to be tested, and `statements` are the lines of code that are to be executed if the condition is met.

Example

```
if (q != 1)
    {a = 0; b = 1;}
```

A.11.2   `if-else` **Statement**

`if-else` statements are another type of conditional control statement. The format of an `if-else` statement is:

```
if (logical-expression)
  {statements}
else
  {statements}
```

where `logical-expression` is the condition to be tested, and the first set of `statements` are the lines of code that are to be executed if the condition is met. If the condition is not met, then the statements following `else` are executed.

Example

```
if (x < 0.)
  y = x/50.;
else
  {
   x = -x;
   y = x/25.;
  }
```

The equivalent FORTRAN code is shown below for comparison.

```
        IF (X.LT.0.) THEN
          Y = X/50.
        ELSE
          X = -X
          Y = X/25.
        ENDIF
```

A.11.3   `for` **Loops**

`for` loops are control statements that are a basic looping construct in C. They are analogous to `do` loops in FORTRAN. The format of a `for` loop is

```
for (begin ; end ; increment)
    {statements}
```

where `begin` is the expression that is executed at the beginning of the loop; `end` is the logical expression that tests for loop termination; and `increment` is the expression that is executed at the end of the loop iteration (usually incrementing a counter).

Example

```
/* Print integers 1-10 and their squares */

int i, j, n = 10;

for (i = 1 ; i <= n ; i++)
   { j = i*i;
     printf("%d %d\n",i,j);
   }
```

The equivalent FORTRAN code is shown below for comparison.

```
INTEGER I,J
N = 10
DO I = 1,10
J = I*I
WRITE (*,*) I,J
ENDDO
```

A.12   Common C Operators

Operators are internal C functions that, when they are applied to values, produce a result. Common types of C operators are arithmetic and logical.

A.12.1   Arithmetic Operators

Some common arithmetic operators are listed below.

```
=  assignment
+  addition
-  subtraction
*  multiplication
/  division
%  modulo reduction
++ increment
-- decrement
```

Note that multiplication, division, and modulo reduction (%) operations will be performed before addition and subtraction in any expression. When division is performed on two integers, the result is an integer with the remainder discarded. Modulo reduction is the remainder from integer division. The ++ operator is a shorthand notation for the increment operation.

A.12.2   Logical Operators

Some common logical operators are listed below.

```
<    less than
<=   less than or equal to
>    greater than
>=   greater than or equal to
==   equal to
!=   not equal to
```

A.13 C Library Functions

C compilers include a library of standard mathematical and I/O functions that you can use when you write your UDF code. Lists of standard C library functions are presented in the following sections. Definitions for standard C library functions can be found in various header files (e.g., `global.h`). These header files are all included in the `udf.h` file.

A.13.1 Trigonometric Functions

The trigonometric functions shown below are computed (with one exception) for the variable x. Both the function and the argument are double-precision `real` variables. The function `acos(x)` is the arccosine of the argument x, $\cos^{-1}(x)$. The function `atan2(x,y)` is the arctangent of `x/y`, $\tan^{-1}(x/y)$. The function `cosh(x)` is the hyperbolic cosine function, etc.

```
double acos (double x);                   returns the arccosine of x
double asin (double x);                   returns the arcsine of x
double atan (double x);                   returns the arctangent of x
double atan2 (double x, double y);        returns the arctangent of x/y
double cos (double x);                    returns the cosine of x
double sin (double x);                    returns the sine of x
double tan (double x);                    returns the tangent of x
double cosh (double x);                   returns the hyperbolic cosine of x
double sinh (double x);                   returns the hyperbolic sine of x
double tanh (double x);                   returns the hyperbolic tangent of x
```

A.13.2 Miscellaneous Mathematical Functions

The C functions shown on the left below correspond to the mathematical functions shown on the right.

```
double sqrt (double x);            √x
double pow(double x, double y);    x^y
double exp (double x);             e^x
double log (double x);             ln(x)
double log10 (double x);           log₁₀(x)
double fabs (double x);            | x |
double ceil (double x);            smallest integer not less than x
double floor (double x);           largest integer not greater than x
```

double `sqrt` (double x); $\sqrt{x}$

double `pow`(double x, double y); x^y

double `exp` (double x); e^x

double `log` (double x); $\ln(x)$

double `log10` (double x); $\log_{10}(x)$

double `fabs` (double x); $|x|$

double `ceil` (double x); smallest integer not less than x

double `floor` (double x); largest integer not greater than x

A.13.3 Standard I/O Functions

A number of standard input and output (I/O) functions are available in C and in ANSYS FLUENT. They are listed below. All of the functions work on a specified file except for `printf`, which displays information that is specified in the argument of the function. The format string argument is the same for `printf`, `fprintf`, and `fscanf`. Note that all of these standard C I/O functions are supported by the interpreter, so you can use them in either interpreted or compiled UDFs. For more information about standard I/O functions in C, you should consult a reference guide (e.g., [6]).

Common C I/O Functions

```
fopen("filename", "mode");      opens a file
fclose(fp);                     closes a file
printf("format", ...);          formatted print to the console
fprintf(fp, "format", ...);     formatted print to a file
fscanf(fp, "format", ...);      formatted read from a file
```

$\boxed{i}$ It is not possible to use the `scanf` C function in ANSYS FLUENT.

fopen

```
FILE *fopen(char *filename, char *mode);
```

The function `fopen` opens a file in the mode that you specify. It takes two arguments: `filename` and `mode`. `filename` is a pointer to the file you want to open. `mode` is the mode in which you want the file opened. The options for `mode` are read `"r"`, write `"w"`, and append `"a"`. Both arguments must be enclosed in quotes. The function returns a pointer to the file that is to be opened.

Before using `fopen`, you will first need to define a local pointer of type `FILE` that is defined in `stdio.h` (e.g., `fp`). Then, you can open the file using `fopen`, and assign it to the local pointer as shown below. Recall that `stdio.h` is included in the `udf.h` file, so you don't have to include it in your function.

```
FILE *fp;    /*  define a local pointer fp of type FILE  */
fp = fopen("data.txt","r");   /*  open a file named data.txt in
                                 read-only mode and assign it to fp */
```

fclose

`int fclose(FILE *fp);`

The function `fclose` closes a file that is pointed to by the local pointer passed as an argument (e.g., `fp`).

```
fclose(fp);   /* close the file pointed to by fp */
```

printf

`int printf(char *format, ...);`

The function `printf` is a general-purpose printing function that prints to the console in a format that you specify. The first argument is the format string. It specifies how the remaining arguments are to be displayed in the console. The format string is defined within quotes. The value of the replacement variables that follow the format string will be substituted in the display for all instances of `%type`. The `%` character is used to designate the character type. Some common format characters are: `%d` for integers, `%f` for floating point numbers, and `%e` for floating point numbers in exponential format (with `e` before the exponent). The format string for `printf` is the same as for `fprintf` and `fscanf`.

In the example below, the text `Content of variable a is:` will be displayed in the console, and the value of the replacement variable, `a`, will be substituted in the message for all instances of `%d`.

Example:

```
int a = 5;
printf("Content of variable a is: %d\n", a); /* \n denotes a new line */
```

> ⓘ (UNIX only) It is recommended that you use the ANSYS FLUENT `Message` utility instead of `printf` for compiled UDFs. See Section 3.7: `Message` for details on the `Message` macro.

fprintf

`int fprintf(FILE *fp, char *format, ...);`

The function `fprintf` writes to a file that is pointed to by `fp`, in a format that you specify. The first argument is the format string. It specifies how the remaining arguments are to be written to the file. The format string for `fprintf` is the same as for `printf` and `fscanf`.

Example:

```
FILE *fp;
fprintf(fp,"%12.4e %12.4e %5d\n",x_array[j][0], x_array[j][1], noface);

int data1 = 64.25;
int data2 = 97.33;
fprintf(fp, "%4.2d %4.2d\n", data1, data2);
```

fscanf

```
int fscanf(FILE *fp, char *format, ...);
```

The function `fscanf` reads from a file that is pointed to by `fp`, in a format that you specify. The first argument is the format string. It specifies how the data that is to be read is to be interpreted. The replacement variables that follow the format string are used to store values that are read. The replacement variables are preceded by the `&` character. Note that the format string for `fscanf` is the same as for `fprintf` and `printf`.

In the example below, two floating point numbers are read from the file pointed to by `fp`, and are stored in the variables `f1` and `f2`.

Example:

```
FILE *fp;
fscanf(fp, "%f %f", &f1, &f2);
```

> **i** You cannot use the `scanf` I/O function in ANSYS FLUENT. You must use `fscanf` instead.

A.14   Preprocessor Directives

The UDF interpreter supports C preprocessor directives including `#define` and `#include`.

Macro Substitution Directive Using `#define`

When you use the `#define` macro substitution directive, the C preprocessor (e.g., `cpp`) performs a simple substitution and expands the occurrence of each argument in *macro* using the *replacement-text.*

`#define`   *macro*   *replacement-text*

For example, the macro substitution directive given by

```
#define RAD 1.2345
```

will cause the preprocessor to replace all instances of the variable `RAD` in your UDF with the number `1.2345`. There may be many references to the variable `RAD` in your function, but you only have to define it once in the macro directive; the preprocessor does the work of performing the substitution throughout your code.

In another example

```
#define AREA_RECTANGLE(X,Y) ((X)*(Y))
```

all of the references to `AREA_RECTANGLE(X,Y)` in you UDF are replaced by the product of `(X)` and `(Y)`.

File Inclusion Directive Using `#include`

When you use the `#include` file inclusion directive, the C preprocessor replaces the line `#include` *filename* with the contents of the named file.

`#include "`*filename*`"`

The file you name must reside in your current folder. The only exception to this rule is the `udf.h` file, which is read automatically by the ANSYS FLUENT solver.

For example, the file inclusion directive given by

```
#include "udf.h"
```

will cause the `udf.h` file to be included with your source code.

The ANSYS FLUENT solver automatically reads the udf.h file from the following folder:

$$path\text{\textbackslash}\texttt{ANSYS Inc\textbackslash v120\textbackslash fluent\textbackslash fluent12.0.}\overset{\Downarrow}{x}\texttt{ \textbackslash src\textbackslash udf.h}$$

where *path* is the folder in which you have installed ANSYS FLUENT (by default, the *path* is `C:\Program Files`), and x is replaced by the appropriate number for the release (e.g., 9 for `fluent12.0.9`). .

A.15 Comparison with FORTRAN

Many simple C functions are similar to FORTRAN function subroutines as shown in the example below:

```
A simple C function        An equivalent FORTRAN function

int myfunction(int x)  INTEGER FUNCTION MYFUNCTION(X)
{
int x,y,z;             INTEGER X,Y,Z
y = 11;                Y = 11
z = x+y;               Z = X+Y
printf("z = %d",z);    WRITE (*,100) Z
return z;              MYFUNCTION = Z
}                      END
```

Appendix B.   `DEFINE` **Macro Definitions**

B.1   General Solver `DEFINE` Macros

The following definitions for general solver `DEFINE` macros (see Section 2.2: General Purpose `DEFINE` Macros) are taken from the `udf.h` header file.

```
#define DEFINE_ADJUST(name, domain) void name(Domain *domain)

#define DEFINE_EXECUTE_AT_END(name) void name(void)

#define DEFINE_EXECUTE_AT_EXIT(name) void name(void)

#define DEFINE_EXECUTE_FROM_GUI(name, libname, mode) \
  void name(char *libname, int mode)

#define DEFINE_EXECUTE_ON_LOADING(name, libname) void name(char *libname)

#define DEFINE_INIT(name, domain) void name(Domain *domain)

#define DEFINE_ON_DEMAND(name) void name(void)

#define DEFINE_RW_FILE(name, fp) void name(FILE *fp)
```

B.2   Model-Specific DEFINE **Macro Definitions**

The following definitions for model-specific DEFINE macros (see Section 2.3: Model-Specific DEFINE Macros) are taken from the udf.h header file.

```
#define DEFINE_CHEM_STEP(name, c, t, p, num_p, n_spe, dt, pres, temp, yk) \
  void name(int cell_t c, Thread *t, Particle *p, int num_p, int n_spe, \
            double *dt, double *pres, double *temp, double *yk)

#define DEFINE_CPHI(name,c,t) \
  real name(cell_t c, Thread *t)

#define DEFINE_DIFFUSIVITY(name, c, t, i) \
  real name(cell_t c, Thread *t, int i)

#define DEFINE_DOM_DIFFUSE_REFLECTIVITY(name ,t, nb, n_a, n_b, diff_ ref_a, \
  diff_tran_a, diff_ref_b, diff_tran_b) \
  void name(Thread *t, int nb,  real n_a, real n_b, real *diff_ref_a, \
            real *diff_tran_a, real *diff_ref_b, real *diff_tran_b)

#define DEFINE_DOM_SPECULAR_REFLECTIVITY(name, f, t, nb, n_a, n_b, \
  ray_direction, e_n, total_internal_reflection, \
  specular_reflectivity, specular_transmissivity) \
  void name(face_t f, Thread *t, int nb,  real n_a, real n_b , \
            real ray_direction[], real e_n[], \
            int *total_internal_reflection, real *specular_reflectivity,\
            real *specular_transmissivity)

#define DEFINE_DOM_SOURCE(name, c, t, ni, nb, emission, in_scattering, \
  abs_coeff, scat_coeff) \
  void name(cell_t c, Thread* t, int ni, int nb, real *emission, \
            real *in_scattering, real *abs_coeff, real *scat_coeff)

#define DEFINE_EMISSIVITY_WEIGHTING_FACTOR(name, c, t, T, nb, \
  emissivity_weighting_factor) \
  void name(cell_t c, Thread* t, real T, int nb, \
            real *emissivity_weighting_factor)

#define DEFINE_GRAY_BAND_ABS_COEFF(name, c, t, nb)  \
   real name(cell_t c, Thread *t, int nb)

#define DEFINE_HEAT_FLUX(name, f, t, c0, t0, cid, cir) \
  void name(face_t f, Thread *t, cell_t c0, Thread *t0, \
            real cid[], real cir[])
```

```
#define DEFINE_NET_REACTION_RATE(name, c, t, particle, pressure, \
  temp, yi, rr, jac) \
  void name(cell_t c, Thread *t, Particle *particle, \
  double *pressure, double *temp, double *yi, double *rr, \
  double *jac)

#define DEFINE_NOX_RATE(name, c, t, Pollut, Pollut_Par, NOx) \
  void name(cell_t c, Thread *t, Pollut_Cell *Pollut, \
  Pollut_Parameter *Poll_Par, NOx_Parameter *NOx)


#define DEFINE_PRANDTL_K(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_D(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_O(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_T(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_T_WALL(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PROFILE(name, t, i) void name(Thread *t, int i)

#define DEFINE_PROPERTY(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PR_RATE(name, c, t, r, mw, ci, p, sf, dif_index, \
  cat_index, rr) \
  void name(cell_t c, Thread *t, Reaction *r, real *mw, real *ci, \
          Tracked_Particle *p, real *sf , int dif_index, \
          int cat_index, real *rr)

#define DEFINE_SCAT_PHASE_FUNC(name, c, f) \
  real name(real c, real *f)

#define DEFINE_SOLAR_INTENSITY(name, sun_x, sun_y, sun_z, S_hour, S_minute) \
   real name(real sun_x, real sun_y, real sun_z, int S_hour,int S_minute)

#define DEFINE_SOURCE(name, c, t, dS, i) \
  real name(cell_t c, Thread *t, real dS[], int i)

#define DEFINE_SOX_RATE(name, c, t, Pollut, Pollut_Par, SOx) \
  void name(cell_t c, Thread *t, Pollut_Cell *Pollut, \
```

```
                  Pollut_Parameter *Poll_Par, SOx_Parameter *SOx)


  #define DEFINE_SR_RATE(name, f, t, r, mw, yi, rr) \
    void name(face_t f, Thread *t, \
      Reaction *r, real *mw, real *yi, real *rr)

  #define DEFINE_TURB_PREMIX_SOURCE(name, c, t, \
    turbulent_flame_speed, source) \
    void name(cell_t c, Thread *t, real *turbulent_flame_speed, \
             real *source)

  #define DEFINE_TURBULENT_VISCOSITY(name, c, t) \
    real name(cell_t c, Thread *t)

  #define DEFINE_VR_RATE(name, c, t, r, mw, yi, rr, rr_t) \
    void name(cell_t c, Thread *t, \
      Reaction *r, real *mw, real *yi, real *rr, real *rr_t)

  #define DEFINE_WALL_FUNCTIONS(name, f, t, c0, t0, wf_ret, yPlus, Emod) \
    real name(face_t f, Thread *t, cell_t c0, Thread *t0, int wf_ret \
             real yPlus, real Emod)
```

B.3   Multiphase DEFINE **Macros**

The following definitions for multiphase DEFINE macros (see Section 2.4: Multiphase DEFINE Macros) are taken from the udf.h header file.

```
  #define DEFINE_CAVITATION_RATE(name, c, t, p, rhoV, rhoL, vofV, p_v, \
    cigma, f_gas, m_dot) \
    void name(cell_t c, Thread *t, real *p, real *rhoV, real *rhoL, \
             real *vofV, real *p_v, real *cigma, real *f_gas, real *m_dot)

  #define DEFINE_EXCHANGE_PROPERTY(name, c, mixture_thread, \
    second_column_phase_index, first_column_phase_index) \
    real name(cell_t c, Thread *mixture_thread, \
             int second_column_phase_index, int first_column_phase_index)

  #define DEFINE_HET_RXN_RATE(name, c, t, hr, mw, yi, rr, rr_t) \
    void name(cell_t c, Thread *t, \
      Hetero_Reaction *hr, real mw[MAX_PHASES][MAX_SPE_EQNS],  \
```

```
              real yi[MAX_PHASES][MAX_SPE_EQNS], real *rr, real *rr_t)


#define DEFINE_MASS_TRANSFER(name, c, mixture_thread, from_phase_index, \
  from_species_index, to_phase_index, to_species_index) \
  real name(cell_t c, Thread *mixture_thread, int from_phase_index, \
  int from_species_index, int to_phase_index, int to_species_index)



#define DEFINE_VECTOR_EXCHANGE_PROPERTY(name, c, mixture_thread, \
  second_column_phase_index, first_column_phase_index, vector_result) \
  void name(cell_t c, Thread *mixture_thread, \
            int second_column_phase_index, \
            int first_column_phase_index, real *vector_result)
```

B.4 Dynamic Mesh Model DEFINE Macros

The following definitions for dynamic mesh model DEFINE macros (see Section 2.6: Dynamic Mesh DEFINE Macros) are taken from the udf.h header file.

```
#define DEFINE_CG_MOTION(name, dt, vel, omega, time, dtime) \
  void name(Dynamic_Thread *dt, real vel[], real omega[], real time,\
            real dtime)

#define DEFINE_DYNAMIC_ZONE_PROPERTY(name, dt, swirl_center) \
  void name(Dynamic_Thread *dt, real *swirl_center)

#define DEFINE_DYNAMIC_ZONE_PROPERTY(name, dt, height) \
  void name(Dynamic_Thread *dt, real *height)

#define DEFINE_GEOM(name, d, dt, position) \
  void name(Domain *d, Dynamic_Thread *dt, real *position)

#define DEFINE_GRID_MOTION(name, d, dt, time, dtime) \
  void name(Domain *d, Dynamic_Thread *dt, real time, real dtime)

#define DEFINE_SDOF_PROPERTIES(name, properties, dt, time, dtime) \
  void name(real *properties, Dynamic_Thread *dt, real time, real dtime)
```

B.5 Discrete Phase Model DEFINE Macros

The following definitions for DPM DEFINE macros (see Section 2.5: Discrete Phase Model (DPM) DEFINE Macros) are taken from the dpm.h header file. Note that dpm.h is included in the udf.h header file.

```
#define DEFINE_DPM_BC(name, p, t, f, normal, dim) \
  int name(Tracked_Particle *p, Thread *t, face_t f, \
           real normal[], int dim)

#define DEFINE_DPM_BODY_FORCE(name, p, i) \
  real name(Tracked_Particle *p, int i)

#define DEFINE_DPM_DRAG(name, Re, p) \
  real name(real Re, Tracked_Particle *p)

#define DEFINE_DPM_EROSION(name, p, t, f, normal, alpha, Vmag, mdot) \
  void name(Tracked_Particle *p, Thread *t, face_t f, real normal[], \
    real alpha, real Vmag, real mdot)

#define DEFINE_DPM_HEAT_MASS(name, p, Cp, hgas, hvap, cvap_surf, dydt, dzdt) \
  void name(Tracked_Particle *p, real Cp, \
           real *hgas, real *hvap, real *cvap_surf, real *dydt, dpms_t *dzdt)

#define DEFINE_DPM_INJECTION_INIT(name, I) void name(Injection *I)

#define DEFINE_DPM_LAW(name, p, ci) \
  void name(Tracked_Particle *p, int ci)

#define DEFINE_DPM_OUTPUT(name, header, fp, p, t, plane) \
  void name(int header, FILE *fp, Tracked_Particle *p, \
           Thread *t, Plane *plane)

#define DEFINE_DPM_PROPERTY(name, c, t, p) \
  real name(cell_t c, Thread *t, Tracked_Particle *p)

#define DEFINE_DPM_SCALAR_UPDATE(name, c, t, initialize, p) \
  void name(cell_t c, Thread *t, int initialize, Tracked_Particle *p)

#define DEFINE_DPM_SOURCE(name, c, t, S, strength, p) \
  void name(cell_t c, Thread *t, dpms_t *S, real strength,\
           Tracked_Particle *p)

#define DEFINE_DPM_SPRAY_COLLIDE(name, tp, p) \
```

```
    void name(Tracked_Particle *tp, Particle *p)

#define DEFINE_DPM_SWITCH(name, p, ci) \
  void name(Tracked_Particle *p, int ci)

#define DEFINE_DPM_TIMESTEP(name, p, ts) \
  real name(Tracked_Particle *p,real ts)

#define DEFINE_DPM_VP_EQUILIB(name, p, cvap_surf) \
  void name(Tracked_Particle *p, real *cvap_surf)
```

B.6   User-Defined Scalar (UDS) DEFINE **Macros**

The following definitions for UDS DEFINE macros (see Section 2.7: User-Defined Scalar (UDS) Transport Equation DEFINE Macros) are taken from the udf.h header file.

```
#define DEFINE_ANISOTROPIC_DIFFUSIVITY(name, c, t, ns, dmatrix) \
    void name(cell_t c, Thread *t, int ns, real dmatrix[ND_ND][ND_ND])

#define DEFINE_UDS_FLUX(name, f, t, i) real name(face_t f, Thread *t, int i)

#define DEFINE_UDS_UNSTEADY(name, c, t, i, apu, su) \
  void name(cell_t c, Thread *t, int i, real *apu, real *su)
```

Appendix C. Quick Reference Guide for Multiphase `DEFINE` **Macros**

This appendix is a reference guide that contains a list of general purpose `DEFINE` macros (Section 2.3: Model-Specific `DEFINE` Macros) and multiphase-specific `DEFINE` macros (Section 2.4: Multiphase `DEFINE` Macros) that can be used to define multiphase model UDFs.

See Section 1.10: Special Considerations for Multiphase UDFs for information on special considerations for multiphase UDFs.

C.1 VOF Model

Tables C.1.1–C.1.3 list the variables that can be customized using UDFs for the VOF multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF needs to be hooked to for the given variable.

Table C.1.1: `DEFINE` Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
Boundary Conditions		
Inlet/Outlet		
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity magnitude	DEFINE_PROFILE	mixture
pressure	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
mass flux	DEFINE_PROFILE	primary and secondary phase(s)
species mass fractions	DEFINE_PROFILE	phase-dependent
internal emissivity	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary condition	DEFINE_PROFILE	mixture

Table C.1.2: DEFINE Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	mixture
energy source	DEFINE_SOURCE	mixture
turbulence kinetic energy source	DEFINE_SOURCE	mixture
turbulence dissipation rate source	DEFINE_SOURCE	mixture
user-defined scalar source	DEFINE_SOURCE	mixture
species source	DEFINE_SOURCE	phase-dependent
velocity	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
user-defined scalar	DEFINE_PROFILE	mixture
turbulence kinetic energy	DEFINE_PROFILE	mixture
turbulence dissipation rate	DEFINE_PROFILE	mixture
species mass fraction	DEFINE_PROFILE	phase-dependent
porosity	DEFINE_PROFILE	mixture
Boundary Conditions **Wall**		
species boundary condition	DEFINE_PROFILE	phase-dependent
internal emissivity	DEFINE_PROFILE	mixture
irradiation	DEFINE_PROFILE	mixture
roughness height	DEFINE_PROFILE	mixture
roughness constant	DEFINE_PROFILE	mixture
shear stress components	DEFINE_PROFILE	mixture
swirl components	DEFINE_PROFILE	mixture
moving velocity components	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
user scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture

Table C.1.3: `DEFINE` Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
Other		
surface tension coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.2   Mixture Model

Tables C.2.1–C.2.3 list the variables that can be customized using UDFs for the Mixture multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF needs to be hooked to for the given variable.

Table C.2.1: `DEFINE` Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
Boundary Conditions		
Inlet/Outlet		
volume fraction	`DEFINE_PROFILE`	secondary phase(s)
mass flux	`DEFINE_PROFILE`	primary and secondary phase(s)
velocity magnitude	`DEFINE_PROFILE`	primary and secondary phase(s)
pressure	`DEFINE_PROFILE`	mixture
temperature	`DEFINE_PROFILE`	mixture
species mass fractions	`DEFINE_PROFILE`	phase-dependent
user-defined scalar boundary value	`DEFINE_PROFILE`	mixture
discrete phase boundary condition	`DEFINE_PROFILE`	mixture

Table C.2.2: DEFINE Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	mixture
energy source	DEFINE_SOURCE	mixture
turbulence kinetic energy source	DEFINE_SOURCE	mixture
turbulence dissipation rate source	DEFINE_SOURCE	mixture
granular temperature source	DEFINE_SOURCE	secondary phase(s)
user scalar source	DEFINE_SOURCE	mixture
species source	DEFINE_SOURCE	phase-dependent
species mass fractions	DEFINE_PROFILE	phase-dependent
velocity	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
turbulence kinetic energy	DEFINE_PROFILE	mixture
turbulence dissipation rate	DEFINE_PROFILE	mixture
porosity	DEFINE_PROFILE	mixture
granular temperature	DEFINE_PROFILE	secondary phase(s)
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
Wall		
roughness height	DEFINE_PROFILE	mixture
roughness constant	DEFINE_PROFILE	mixture
internal emissivity	DEFINE_PROFILE	mixture
shear stress components	DEFINE_PROFILE	mixture
moving velocity components	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
user scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
species boundary condition	DEFINE_PROFILE	phase-dependent

Table C.2.3: `DEFINE` Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
Material Properties		
cavitation surface tension coefficient	`DEFINE_PROPERTY`	phase interaction
cavitation vaporization pressure	`DEFINE_PROPERTY`	phase interaction
particle or droplet diameter	`DEFINE_PROPERTY`	secondary phase(s)
granular diameter	`DEFINE_PROPERTY`	secondary phase(s)
granular solids pressure	`DEFINE_PROPERTY`	secondary phase(s)
granular radial distribution	`DEFINE_PROPERTY`	secondary phase(s)
granular elasticity modulus	`DEFINE_PROPERTY`	secondary phase(s)
granular viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular temperature	`DEFINE_PROPERTY`	secondary phase(s)
Other		
slip velocity	`DEFINE_VECTOR_EXCHANGE_PROPERTY`	phase interaction
drag coefficient	`DEFINE_EXCHANGE`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.3   Eulerian Model - Laminar Flow

Tables C.3.1–C.3.3 list the variables that can be customized using UDFs for the laminar flow Eulerian multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF needs to be hooked to for the given variable.

Table C.3.1: DEFINE Macro Usage for the Eulerian Model - Laminar Flow

Variable	Macro	Phase Specified On
Boundary Conditions		
Inlet/Outlet		
volume fraction	DEFINE_PROFILE	secondary phase(s)
species mass fractions	DEFINE_PROFILE	phase-dependent
mass flux	DEFINE_PROFILE	primary and secondary phase(s)
flow direction components	DEFINE_PROFILE	primary and secondary phase(s)
velocity magnitude	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
pressure	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
species source	DEFINE_SOURCE	phase-dependent
granular temperature source	DEFINE_SOURCE	secondary phase(s)
user-defined scalar source	DEFINE_SOURCE	mixture
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)

Table C.3.2: `DEFINE` Macro Usage for the Eulerian Model - Laminar Flow

Variable	Macro	Phase Specified On
Boundary Conditions		
Fluid		
species mass fraction	DEFINE_PROFILE	phase-dependent
granular temperature	DEFINE_PROFILE	secondary phase(s)
porosity	DEFINE_PROFILE	mixture
user-defined scalar	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
Wall		
species boundary condition	DEFINE_PROFILE	phase-dependent
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	secondary phase(s)
temperature	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Material Properties		
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular solids pressure	DEFINE_PROPERTY	secondary phase(s)
granular radial distribution	DEFINE_PROPERTY	secondary phase(s)
granular elasticity modulus	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular temperature	DEFINE_PROPERTY	secondary phase(s)

Table C.3.3: `DEFINE` Macro Usage for the Eulerian Model - Laminar Flow

Variable	Macro	Phase Specified On
Other		
drag coefficient	`DEFINE_EXCHANGE`	phase interaction
lift coefficient	`DEFINE_EXCHANGE`	phase interaction
heat transfer coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.4   Eulerian Model - Mixture Turbulence Flow

Tables C.4.1–C.4.3 list the variables that can be customized using UDFs for the mixed turbulence flow Eulerian multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF needs to be hooked to for the given variable.

Table C.4.1: `DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions		
Inlet/Outlet		
volume fraction	`DEFINE_PROFILE`	secondary phase(s)
species mass fractions	`DEFINE_PROFILE`	phase-dependent
mass flux	`DEFINE_PROFILE`	primary and secondary phases(s)
velocity magnitude	`DEFINE_PROFILE`	primary and secondary phases(s)
temperature	`DEFINE_PROFILE`	primary and secondary phases(s)
pressure	`DEFINE_PROFILE`	mixture

Table C.4.2: `DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet - continued		
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary condition	DEFINE_PROFILE	mixture
Fluid mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence dissipation rate source	DEFINE_SOURCE	mixture
turbulence kinetic energy source	DEFINE_SOURCE	mixture
user-defined scalar source	DEFINE_SOURCE	mixture
user-defined scalar	DEFINE_PROFILE	mixture
turbulence kinetic energy	DEFINE_PROFILE	mixture
turbulence dissipation rate	DEFINE_PROFILE	mixture
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
porosity	DEFINE_PROFILE	mixture
user-defined scalar	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
Wall		
species boundary condition	DEFINE_PROFILE	phase-dependent
shear stress components	DEFINE_PROFILE	primary and secondary phase(s
moving velocity components	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture

Table C.4.3: `DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow

Variable	Macro	Phase Specified On
Wall - continued		
granular flux	`DEFINE_PROFILE`	secondary phase(s)
granular temperature	`DEFINE_PROFILE`	secondary phase(s)
discrete phase boundary condition	`DEFINE_DPM_BC`	secondary phase(s)
user-defined scalar boundary value	`DEFINE_PROFILE`	secondary phase(s)
Material Properties		
granular diameter	`DEFINE_PROPERTY`	secondary phase(s)
granular viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular bulk viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular frictional viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular conductivity	`DEFINE_PROPERTY`	secondary phase(s)
granular solids pressure	`DEFINE_PROPERTY`	secondary phase(s)
granular radial distribution	`DEFINE_PROPERTY`	secondary phase(s)
granular elasticity modulus	`DEFINE_PROPERTY`	secondary phase(s)
turbulent viscosity	`DEFINE_TURBULENT_VISCOSITY`	mixture, primary, and secondary phase(s)
Other		
drag coefficient	`DEFINE_EXCHANGE`	phase interaction
lift coefficient	`DEFINE_EXCHANGE`	phase interaction
heat transfer coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.5   Eulerian Model - Dispersed Turbulence Flow

Tables C.5.1–C.5.3 list the variables that can be customized using UDFs for the dispersed turbulence flow Eulerian multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF needs to be hooked to for the given variable.

Table C.5.1: `DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions **Inlet/Outlet**		
volume fraction	DEFINE_PROFILE	secondary phase(s)
species mass fractions	DEFINE_PROFILE	phase-dependent
mass flux	DEFINE_PROFILE	primary and secondary phases(s)
velocity magnitude	DEFINE_PROFILE	primary and secondary phases(s)
temperature	DEFINE_PROFILE	primary and secondary phases(s)
pressure	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary condition	DEFINE_PROFILE	mixture
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence dissipation rate source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence kinetic energy source	DEFINE_SOURCE	primary and secondary phase(s)
species source	DEFINE_SOURCE	phase-dependent
user-defined scalar source	DEFINE_SOURCE	mixture
turbulence dissipation rate	DEFINE_PROFILE	primary and secondary phase(s)
turbulence kinetic energy	DEFINE_PROFILE	primary and secondary phase(s)

Table C.5.2: DEFINE Macro Usage for the Eulerian Model - Dispersed Turbulence Flow

Variable	Macro	Phase Specified On
Fluid		
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
species mass fraction	DEFINE_PROFILE	primary and secondary phase(s)
porosity	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
user-defined scalar	DEFINE_PROFILE	mixture
Wall		
species mass fraction	DEFINE_PROFILE	mixture
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Material Properties		
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)
conductivity	DEFINE_PROPERTY	secondary phase(s)
granular solids pressure	DEFINE_PROPERTY	secondary phase(s)
granular radial distribution	DEFINE_PROPERTY	secondary phase(s)
granular elasticity modulus	DEFINE_PROPERTY	secondary phase(s)
turbulent viscosity	DEFINE_TURBULENT_ VISCOSITY	mixture, primary, and secondary phase(s)

Table C.5.3: `DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow

Variable	Macro	Phase Specified On
Other		
drag coefficient	`DEFINE_EXCHANGE`	phase interaction
lift coefficient	`DEFINE_EXCHANGE`	phase interaction
heat transfer coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.6   Eulerian Model - Per Phase Turbulence Flow

Tables C.6.1–C.6.3 list the variables that can be customized using UDFs for the per phase turbulence flow Eulerian multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF needs to be hooked to for the given variable.

Table C.6.1: `DEFINE` Macro Usage for the Eulerian Model - Per Phase Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions		
Inlet/Outlet		
volume fraction	`DEFINE_PROFILE`	secondary phase(s)
species mass fractions	`DEFINE_PROFILE`	phase-dependent
mass flux	`DEFINE_PROFILE`	primary and secondary phases(s)
velocity magnitude	`DEFINE_PROFILE`	primary and secondary phases(s)
temperature	`DEFINE_PROFILE`	primary and secondary phases(s)
pressure	`DEFINE_PROFILE`	mixture
user-defined scalar boundary value	`DEFINE_PROFILE`	mixture

Table C.6.2: DEFINE Macro Usage for the Eulerian Model - Per Phase Turbulence Flow

Variable	Macro	Phase Specified On
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence dissipation rate source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence kinetic energy source	DEFINE_SOURCE	primary and secondary phase(s)
user-defined scalar source	DEFINE_SOURCE	mixture
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
turbulence kinetic energy	DEFINE_PROFILE	primary and secondary phase(s)
turbulence dissipation rate	DEFINE_PROFILE	primary and secondary phase(s)
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
porosity	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
user-defined scalar	DEFINE_PROFILE	mixture

Table C.6.3: `DEFINE` Macro Usage for the Eulerian Model - Per Phase Turbulence Flow

Variable	Macro	Phase Specified On
Wall		
species boundary condition	DEFINE_PROFILE	phase-dependent
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Material Properties		
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)
granular conductivity	DEFINE_PROPERTY	secondary phase(s)
granular solids pressure	DEFINE_PROPERTY	secondary phase(s)
granular radial distribution	DEFINE_PROPERTY	secondary phase(s)
granular elasticity modulus	DEFINE_PROPERTY	secondary phase(s)
turbulent viscosity	DEFINE_TURBULENT_ VISCOSITY	mixture, primary, and secondary phase(s)
Other		
drag coefficient	DEFINE_EXCHANGE	phase interaction
lift coefficient	DEFINE_EXCHANGE	phase interaction
heat transfer coefficient	DEFINE_PROPERTY	phase interaction
mass transfer coefficient	DEFINE_MASS_TRANSFER	phase interaction
heterogeneous reaction rate	DEFINE_HET_RXN_RATE	phase interaction

Bibliography

[1] R. H. Aungier. A Fast, Accurate Real Gas Equation of State for Fluid Dynamic Analysis Applications. *Journal of Fluids Engineering*, 117:277–281, 1995.

[2] N. P. Cheremisinoff. *Fluid Flow Pocket Handbook*. Gulf Publishing Co., Houston, TX., 1984.

[3] A.M. Douaud and P. Eyzat. Four-Octane-Number Method for Predicting the Anti-Knock Behavior of Fuels in Engines. SAE Transactions 780080, 1978.

[4] E. R. G. Eckert and R. M. Drake. *Analysis of Heat and Mass Transfer*. McGraw-Hill Co., 1972.

[5] S. Jendoubi, H. S. Lee, and T. K. Kim. Discrete Ordinates Solutions for Radiatively Participating Media in a Cylindrical Enclosure. *J. Thermophys. Heat Transfer*, 7(2):213–219, 1993.

[6] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[7] J. C. Livengood and P. C. Wu. Correlation of Autoignition Phenomena in Internal Combustion Engines and Rapid Compression Machines. In *Fifth Symposium (Int.) on Combustion*, pages 347–356, 1955.

[8] M. J. Moran and H. N. Shapiro. *Fundamentals of Engineering Thermodynamics*. John Wiley & Sons, Inc, 1988.

[9] S. Oualline. *Practical C Programming*. O'Reilly Press, 1997.

[10] M. R. Speigel. *Mathematical Handbook of Formulas and Tables*. Shaum's Outline Series, McGraw-Hill Co, 1968.

Index